



绿盟科技官方微信



2018 绿盟科技 容器安全技术报告

绿盟科技星云实验室



关于绿盟科技

北京神州绿盟信息安全科技股份有限公司（以下简称绿盟科技），成立于 2000 年 4 月，总部位于北京。在国内外设有 40 多个分支机构，为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。

基于多年的安全攻防研究，绿盟科技在检测防御类、安全评估类、安全平台类、远程安全运维服务、安全 SaaS 服务等领域，为客户提供入侵检测 / 防护、抗拒绝服务攻击、远程安全评估以及 Web 安全防护等产品以及安全运营等专业安全服务。

北京神州绿盟信息安全科技股份有限公司于 2014 年 1 月 29 日起在深圳证券交易所创业板上市交易，股票简称：绿盟科技，股票代码：300369。



关于 NeuVector

NeuVector 是最早开发 Docker/Kubernetes 安全产品的公司，是 Kubernetes 网络安全的领导者，是第一也是目前唯一的多矢量容器安全平台发明者。

NeuVector 致力于保障企业级容器平台安全，产品适用于各种云环境、跨云或者本地部署等容器生产环境。NeuVector 提供实时深入的容器网络可视化、东西向容器网络监控、主动隔离和保护、容器主机安全以及容器内部安全。和容器管理平台无缝集成并且实现应用级容器安全的自动化。

NeuVector 的客户涵盖了全球各个领域的大型领导企业：金融行业、医疗保险行业、出版业、新兴的互联网企业等等。NeuVector 的技术以及商业合作伙伴包括：AWS、Docker、Google、IBM、Rancher、Red Hat、阿里云等等。NeuVector 拥有丰富经验的创始团队，来自于各个硅谷的知名公司，包括 Fortinet、VMware、Trend Micro、Symantec、Juniper 等等。NeuVector 在行为学习、网络安全、数据安全、容器安全等领域拥有多项美国技术专利。



2018 绿盟科技容器安全技术报告



绿盟科技星云实验室
2018 年 10 月

C O N T A I N E R   S E C U R I T Y

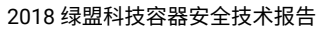


目录

前言	1
1. 概述	2
1.1 容器与虚拟化	3
1.2 容器发展历程	4
1.3 容器安全	5
2. 容器基础	8
2.1 容器镜像	9
2.1.1 镜像简介	9
2.1.2 镜像特点	9
2.1.3 镜像的构建	10
2.1.4 镜像仓库	11
2.1.5 镜像的使用	12
2.2 容器存储	13
2.2.1 镜像元数据	13
2.2.2 存储驱动	13
2.2.3 数据卷	14
2.3 容器网络	15
2.3.1 容器网络支撑技术	15
2.3.2 主机网络	16
2.3.3 集群网络	18
2.4 容器管理与应用	22
2.4.1 容器管理	22
2.4.2 容器使用场景	28
3. 安全风险和挑战	32
3.1 脆弱性和安全风险分析	33
3.1.1 软件风险	33
3.1.2 API 接口安全	35
3.1.3 不安全的镜像	40
3.1.4 容器隔离失效	41



3.2 安全威胁分析.....	41
3.2.1 容器逃逸攻击.....	41
3.2.2 容器网络攻击.....	41
3.2.3 拒绝服务攻击.....	42
3.3 容器应用安全威胁.....	43
3.3.1 微服务安全.....	43
3.3.2 DevOps 安全.....	43
4. 安全容器防护.....	44
4.1 Linux 内核安全机制.....	45
4.1.1 内核命名空间.....	45
4.1.2 控制组.....	45
4.1.3 内核功能限制（Capabilities）机制.....	45
4.1.4 其它内核安全机制.....	46
4.2 容器服务安全.....	47
4.3 主机安全.....	48
4.3.1 主机基本安全加固建议.....	48
4.3.2 容器相关安全加固建议.....	49
4.4 镜像安全.....	50
4.4.1 镜像构建安全.....	50
4.4.2 镜像仓库安全.....	51
4.4.3 镜像扫描.....	51
4.4.4 镜像传输安全.....	52
4.5 容器网络安全.....	52
4.5.1 网络自身安全机制.....	52
4.5.2 容器网络安全防护.....	60
4.6 运行时安全.....	63
4.6.1 容器启动安全配置.....	63
4.6.2 运行时安全监控与审计.....	67
4.7 编排安全.....	71
4.7.1 Kubernetes 安全防护.....	71
4.8 应用安全.....	74
4.8.1 微服务安全.....	74
4.8.2 DevOps 安全.....	75



特别声明

为避免合作伙伴及客户数据泄露，所有数据在进行分析前都已经过匿名化处理，不会在中间环节出现泄露，任何与客户有关的具体信息，均不会出现在本报告中。



前言

容器技术通过共享主机操作系统内核，实现轻量的资源虚拟化和隔离，近年来在 DevOps、微服务等领域有着广泛的应用。

然而在容器技术被广泛接受和使用的同时，容器以及容器运行环境的安全成为了亟待研究和解决的问题。为了进一步了解容器以及容器环境的安全威胁，为使用容器的用户提供安全防护建议。北京神州绿盟信息安全科技股份有限公司（以下简称“绿盟科技”）携手硅谷知名容器安全公司 NeuVector，联合发布《2018 绿盟科技容器安全技术报告》。

本报告主要包括五个章节。

第一章，分别从容器和虚拟化技术的对比、容器技术的发展历史、容器安全问题等几个方面，对全文进行简要概述。

第二章，主要介绍了容器的基本技术原理。包括容器镜像、容器的存储、容器以及容器集群的网络、容器的管理编排平台、容器的应用等几个方面。

第三章，分别从容器的软件实现、API 设计、容器镜像、容器应用等几个方面，具体阐述了容器以及容器运行环境存在的安全威胁。

第四章，针对前述的安全威胁，本章从 Linux 内核安全机制、容器主机安全、容器镜像安全、容器网络安全以及容器运行时安全等几个方面，详细分析介绍了相应的威胁检测以及安全防护方法。

第五章，主要介绍了一些容器安全的工具和产品，包括 Clair、Grafec 等开源插件工具，以及 NeuVector 和 StackRox 等商业公司标准产品。

本报告在编写过程中参考了大量资料，吸取了多方的宝贵意见和建议，在此深表感谢。报告的编写和发布得到了相关单位的大力支持，我们在此表示衷心的感谢！欢迎广大读者批评、指正。



2018 绿 盟 科 技
容器安全技术报告

1. 概述

1.1 容器与虚拟化.....	3
1.2 容器发展历程.....	4
1.3 容器安全.....	5

近年来，云计算的模式逐渐被业界认可和接受。在国内，包括政府、金融、运营商、能源等众多行业，以及中小企业，均将其业务进行不同程度的云化。但简单地将主机、平台或应用转为虚拟化形态，并不能解决传统应用的升级缓慢、架构臃肿、无法快速迭代等问题，于是云原生（Cloud Native）的概念应运而生。

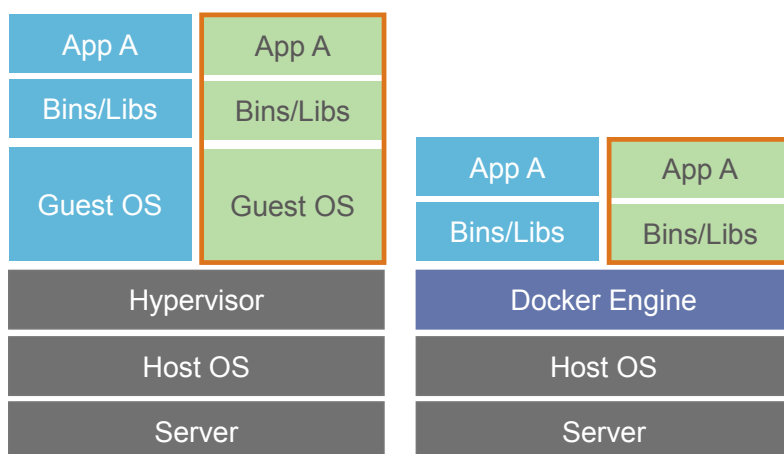
云原生提倡应用的敏捷、可靠、高弹性、易扩展以及持续的更新。在云原生应用和服务平台构建过程中，近年兴起的容器技术凭借其弹性敏捷的特性和活跃强大的社区支持，成为了云原生等应用场景下的重要支撑技术。本章主要介绍容器技术和容器安全的基础概念。

1.1 容器与虚拟化

虚拟化（Virtualization）和容器（Container）都是系统虚拟化的实现技术，可实现系统资源的“一虚多”共享。容器技术是一种“轻量”的虚拟化方式，此处的“轻量”主要是相比于虚拟化技术而言的。例如，虚拟化通常在 Hypervisor 层实现对硬件资源的虚拟化，Hypervisor 为虚拟机提供了虚拟的运行平台，管理虚拟机的操作系统运行，每个虚拟机都有自己的操作系统、系统库以及应用。而容器并没有 Hypervisor 层，每个容器是和主机共享硬件资源及操作系统¹。

容器技术在操作系统层面实现了对计算机系统资源的虚拟化，在操作系统中，通过对 CPU、内存和文件系统等资源的隔离、划分和控制，实现进程之间透明的资源使用。图 1.1 展示了虚拟机和容器在实现架构上的区别。

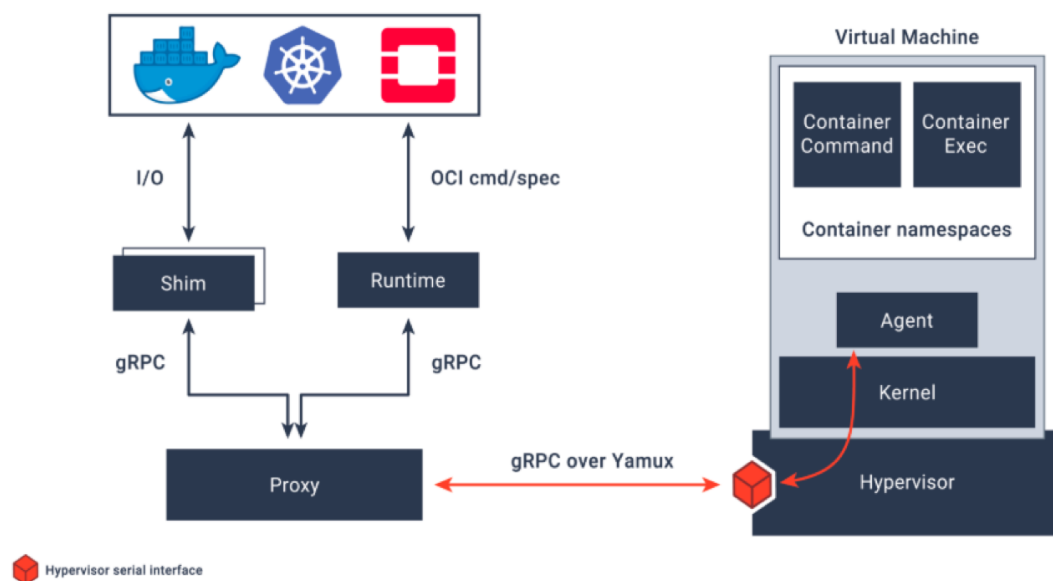
图 1.1 虚拟机和容器架构对比



¹ 本文中，“宿主机”、“容器主机”、“主机”、“Host”均代表容器运行所依托的操作系统内核所运行的主机，既可以是物理服务器，也可以是云环境中的虚拟机。

随着技术的发展和实际业务中对隔离性、性能等指标的平衡，容器和虚拟化两种技术也有融合的趋势，例如 Kata Containers^[1] 项目在轻量虚拟机中内嵌容器，获得了容器级别的启动和运行速度，以及虚拟机级别的隔离和安全性。

图 1.2 Kata Containers 架构



1.2 容器发展历程

容器的概念最早可以追溯到 1979 年的 Unix 工具 Chroot，2000 年左右，FreeBSD 引入的 Jails 算是早期的容器技术之一，2004 年 Solaris 提出 Container，引入了容器资源管理的概念。

2008 年出现的 LXC（Linux Containers）可以说是第一个完整的 Linux 容器管理实现方案，它通过 Linux CGroups (Control Groups) 以及 Linux Namespace 技术实现，LXC 存在于 liblxc 库中，提供各种编程语言的 API 实现，无需任何额外的补丁就能够运行在原版 Linux 内核上。

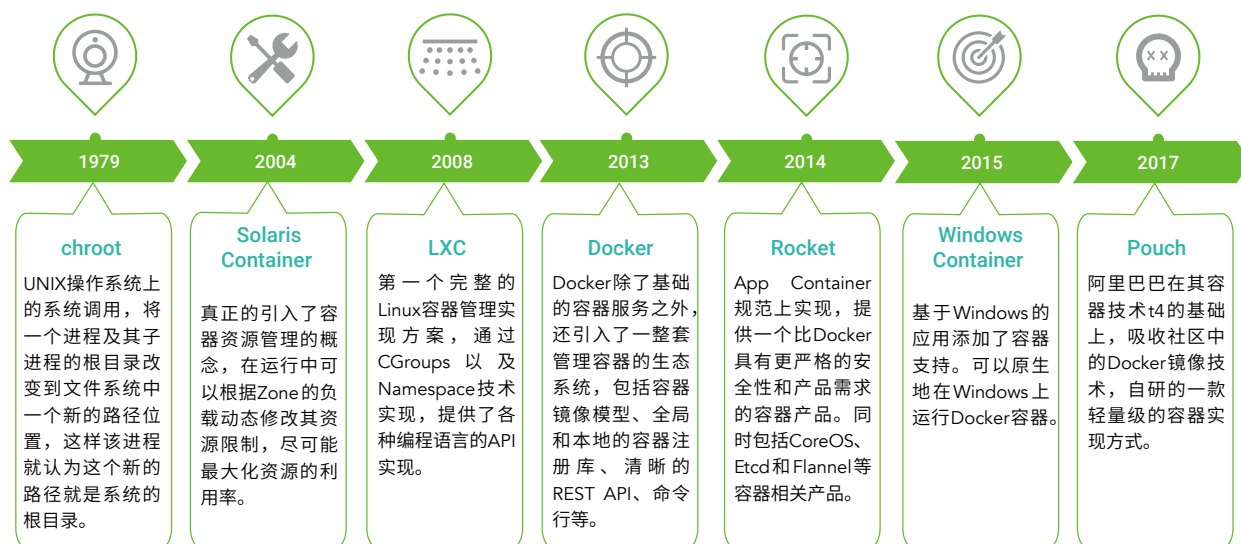
2013 年，DotCloud 开源了其内部的容器项目 Docker^[2]。Docker 在开始阶段是基于 LXC 技术，之后则采用自己开发的 libcontainer 进行了替换。Docker 除了基础的容器服务之外，还引入了一整套管理容器的生态系统，包括容器镜像模型、镜像仓库、REST API、命令行等。

2014 年，CoreOS 发布了容器引擎 Rocket^[3]（简称 rkt），它是在一个更加开放的标准 App Container 上实现的。除 Rocket 之外，CoreOS 也开发了其它几个容器相关的产品，如 CoreOS 操作系统、分布式键值存储组件 Etcd 和网络组件 Flannel。

2015 年，微软也在其 Windows Server 上为基于 Windows 的应用添加了容器支持，称之为 Windows Containers^[4]。该实现可以在 Windows 上原生地运行 Docker 容器。

2017 年 11 月，阿里巴巴开源了其基于 Apache 2.0 许可协议的轻量级容器技术 Pouch^[5]，Pouch 在其容器技术 t4 的基础上，逐渐吸收了社区中的 Docker 镜像技术，具有快速高效、可移植性高、资源占用少等特性，目前在阿里的使用场景中已经扮演了重要角色。

图 1.3 容器技术发展历程



1.3 容器安全

虽然容器技术备受追捧，在多个领域得到广泛使用，但其背后的安全问题不容忽视。

2018 年 6 月 14 日，某安全厂商发现 17 个受到感染的 Docker 容器镜像，镜像中包含了可用于挖掘加密货币的程序，更危险的是，这些镜像的下载次数已经高达 500 万次^[6]。

云安全厂商 Lacework 在 6 月 18 日发布的一份研究报告^[7]中指出，超过 21,000 个容器编排平台目前暴露在互联网上，其中包括 Kubernetes、Docker Swarm、Mesos Marathon、Redhat OpenShift 等被广泛使用的容器编排平台，有将近 3/4 的暴露平台为 Kubernetes（详见 3.1.2 节绿盟威胁情报中心数据）。其中的 305 个甚至都没有设置登录密码，一旦被恶意操作，后果将不堪设想。

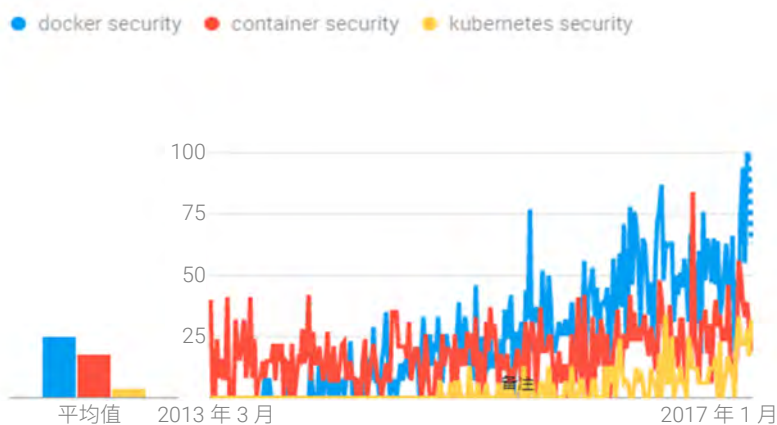
首先，从对业界最流行的容器实现技术 Docker 分析来看，Docker 自 2014 年正式发布以来，截止到 2018 年 7 月 31 日，累计共发现 38 个相关漏洞^[8]，其中 2018 年 8 个，2017 年 5 个，2016 年 11 个。所有 38 个漏洞中，标记危害等级的漏洞有 33 个，其中超危漏洞 4 个，高危漏洞 8 个，仅有 3 个为低危漏洞。

例如 2014 年年底发现的漏洞编号为 CVE-2014-9357 的漏洞，被 CVSS V2 认定为综合评分 10 分的超危漏洞。漏洞造成的风险主要包括被拒绝服务、访问控制权限的绕过、执行任意代码等。

Docker 公司在 2017 年 3 月份宣布正式发布企业版 Docker EE (Docker Enterprise Edition)，版本编号也从 1.13.X 跳跃到了 17.X (17.03.0-ee-1/17.03.1-ee-2、17.03.0-ce/17.03.1-ce)。无论是社区版还是企业版，Docker 在最近的几次版本更新中，均在很大程度上增加了安全性考虑。

例如在 2017 年 11 月 27 日发布的 Docker EE 17.06.2-ee-6^[9] 版本，修复了漏洞容器的执行状态 (moby/moby#35484^[10])，同时增加了对健康状态监测通道进行保护 (moby/moby#35482^[11])，修复了异常镜像可能导致守护进程崩溃，造成内存耗尽的问题 (moby/moby#35424^[12])。

图 1.4 Google Trends 关键字热度趋势



全球过去 5 年 网页搜索

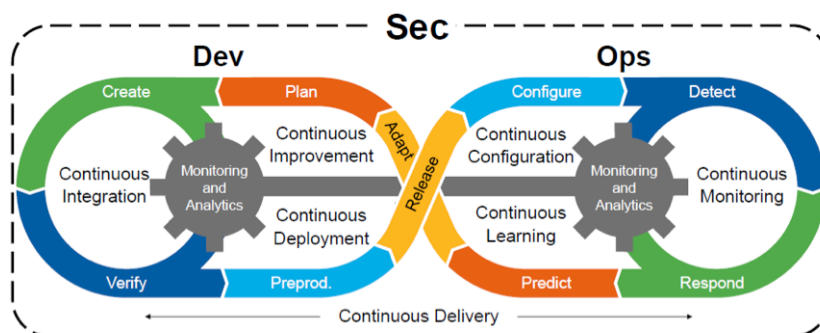
此外，我们通过在 Google Trends¹ 中，比较 “docker security”、“container security” 和 “kubernetes security” 这三个关键词近 5 年时间内，在 Google 的网页搜索热度。从图 1.4 可以看出²，三个关键词的搜索热度均不同程度地呈现出上升趋势，尤其是 “docker security” 在近一两年的热度评分均在 50 分以上，在 2018 年 2 月 4 日—10 日期间达到了 100 分。

1 Google Trends (Google 搜索趋势)，是 Google 开发的一款服务，用于分析用户在 Google 中搜索过的条目。Google Trends 可以对数个不同条目的搜索行为进行比较，也可以针对一个条目在不同地区和时间上的搜索行为进行比较。

2 纵坐标数字代表相对于图表中指定区域和指定时间内最高点的搜索热度。热度最高的字 / 词得 100 分；热度是前者一半的字 / 词得 50 分；没有足够数据的字 / 词得 0 分。

Gartner 在 2017 年 6 月发布了《2017 年度最新最酷的信息安全技术》^[13]，其中就包括容器安全（Container Security）以及 DevOps 安全——DevSecOps（Development -Security-Operations）。在 Gartner 发布的《2017 年云安全技术成熟度曲线》^[14]（Hype Cycle）中，容器安全目前处于新兴（Innovation Trigger）阶段，虽然尚未成熟，但技术进步较快。容器自身安全，以及容器生态环境中的安全防护，都是准备部署容器的企业中 CIO 和 CISO 们非常关心的安全问题。

图 1.5 DevSecOps 能力闭环



一方面，容器技术应用于 DevOps 的开发、测试和运维环节，需要保证全生命周期的安全，如开源库和容器镜像安全评估、编排安全和容器运行时安全等方面，都是 DevSecOps 需要解决的问题；另一方面，在实际的运行环境中，容器自身的系统、应用和网络也需要做安全加固、安全检测和安全防护。

容器本质上是一种操作系统层面的虚拟化技术，一旦攻击者利用宿主机操作系统的某个内核漏洞对容器进行攻击，可能逃逸到宿主机上，并将导致宿主机上的其它容器被攻陷。另外，容器系统本身也存在一定不安全性，使用者在部署和使用容器的时候，往往又缺少专业安全团队的帮助，很容易给攻击者造成可乘之机。

由于容器是介于基础设施和平台之间的虚拟化技术，因此面向基础设施虚拟化的传统云安全解决方案无法完全解决前述安全问题。如以容器为支撑技术构建 DevOps 环境，就需要设计涵盖从容器镜像的创建到投产上线的整个生命周期的容器安全方案。

综上所述，容器以及容器运行环境的安全问题亟待调研和分析，了解容器技术和基于容器技术的系统存在的风险、应对策略，将是构建安全的云原生环境非常重要的前提。



2. 容器基础

2.1 容器镜像.....	9
2.2 容器存储.....	13
2.3 容器网络.....	15
2.4 容器管理与应用.....	22

2.1 容器镜像

镜像是容器运行的基础，容器引擎服务可使用不同的镜像启动相应的容器。在容器出现错误后，能迅速通过删除容器、启动新的容器来恢复服务，这都需要以容器镜像作为支撑技术^[15]。

2.1.1 镜像简介

镜像是由按层封装好的文件系统和描述镜像的元数据构成的文件系统包，包含应用所需要的系统、环境、配置和应用本身等。镜像由开发者构建好之后上传至镜像仓库，使用者获取镜像之后就可以使用镜像直接构建自己的应用。

由 Linux 基金会主导开发的开放容器标准规范（Open Container Initiative, OCI）于 2017 年发布 v1.0 版本^[16]，该标准将致力于统一容器运行时和镜像格式的规范。Docker 积极为 OCI 做出重要贡献，开发并捐赠了大部分的 OCI 代码，并作为项目维护者在定义运行时和镜像规范时做了建设性工作。

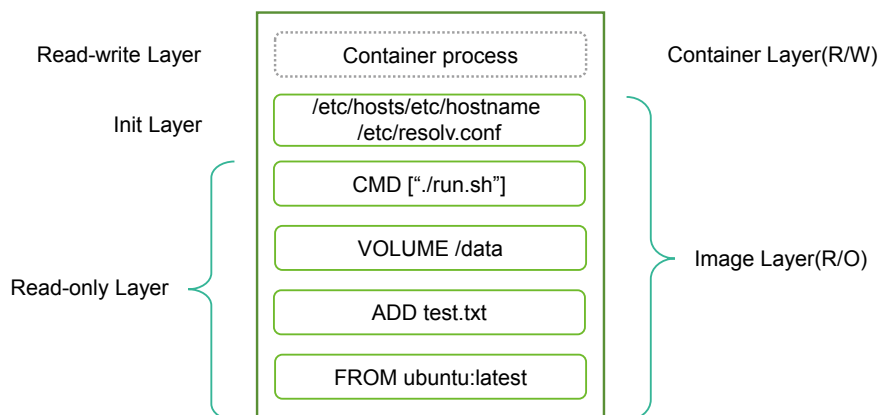
与虚拟机所用的系统镜像不同，容器镜像不仅没有 Linux 系统内核，同时在格式上也有很大的区别。虚拟机镜像是将一个完整系统封装成一个镜像文件，而容器镜像不是一个文件，而是分层存储的文件系统。

2.1.2 镜像特点

（1）分层存储

分层存储是容器镜像的主要特点之一，从图 2.1 可以看出，每个镜像都是由一系列的“镜像层”组成。当需要修改镜像内的某个文件时，只会对最上方的读写层进行改动，不会覆盖下层已有文件系统的内容。当提交这个修改生成新的镜像时，保存的内容仅为最上层可读写文件系统中被更新过的文件，这样就实现了在不同的容器镜像间共享镜像层的效果。下图是一个简单的容器镜像示例，最上层是容器的读写层，剩余的是只读层。

图 2.1 容器镜像结构



(2) 写时复制

容器镜像使用了写时复制 (Copy-on-Write) 的策略, 在多个容器之间共享镜像, 每个容器在启动的时候并不需要单独复制一份镜像文件, 而是将所有镜像层以只读的方式挂载到一个挂载点, 再在上面覆盖一个可读写的容器层。在未更改文件内容时, 所有容器共享同一份数据, 只有在容器运行过程中文件系统发生变化时, 才会把变化的文件内容写到可读写层, 并隐藏只读层中的老版本文件。写时复制配合分层机制减少了镜像对磁盘空间的占用和容器启动时间。

(3) 内容寻址

在 Docker 1.10 版本后, 引入了内容寻址存储的机制, 根据文件的内容来索引镜像和镜像层。对镜像层的内容计算校验和, 生成一个内容哈希值, 并以此哈希值作为镜像层的唯一标识。该机制提高了镜像的安全性, 并在 *pull*、*push*、*load* 和 *save* 操作后检测数据的完整性。

(4) 联合挂载

联合挂载技术可以在一个挂载点上同时挂载多个文件系统, 将挂载点的原目录与被挂载内容进行整合, 使得最终可见的文件系统包含整合之后各层的文件和目录。实现这种联合挂载技术的文件系统通常被称为联合文件系统 (Union Filesystem)。

联合挂载是用于将多个镜像层的文件系统挂载到一个挂载点来实现一个统一文件系统视图的途径, 是下层存储驱动实现分层合并的方式。

2.1.3 镜像的构建

镜像作为容器运行的基础, 有多种获取途径。其中一种是在镜像仓库中获取现成的镜像, 包括公共仓库和私有仓库; 另一种是开发人员自己打包制作镜像。本节主要介绍两种如何制作并生成镜像的方法。

(1) docker build

这种构建方法基于 Dockerfile 来自动构建镜像, Dockerfile 的可读性和可理解性都较高。其机制为: 每一行都会基于上一层的中间容器来执行对应的修改命令, 然后通过 *docker build* 提交, 经过一次次循环, 最终提交成为目标镜像。构建镜像的 Dockerfile 示例如下:

```
FROM ubuntu:15.04
COPY . /app
RUN make /app
CMD python /app/app.py
```

Dockerfile 中的每一行指令 (*COPY*、*RUN*、*CMD*) 都会生成新的一层, 叠加在上一个指令生成的文件系统之上, 最后所有镜像层叠加就构成了镜像的文件系统。

(2) docker commit

这种构建镜像的方式, 首先使用某一镜像启动容器, 进入容器中完成需要的操作, 最后在宿主机上执行 *docker commit* 命令, 该命令会将此时的容器打包成一个新的镜像。这种制作新镜像的优点是修改便捷、构建的镜像出现问题时容易排查解决。缺点在于不够透明、可维护性差。

2.1.4 镜像仓库

镜像仓库是镜像存储的位置，也是用来获得镜像的重要渠道之一。镜像仓库根据用途不同分为公共仓库和私有仓库。

(1) 公共仓库

公共仓库是面向整个互联网用户的仓库，典型的代表为 Docker Hub^[17]，目前已经包括了超过 15,000 的镜像。大部分常用应用的镜像，都可以通过在 Docker Hub 中直接下载使用。

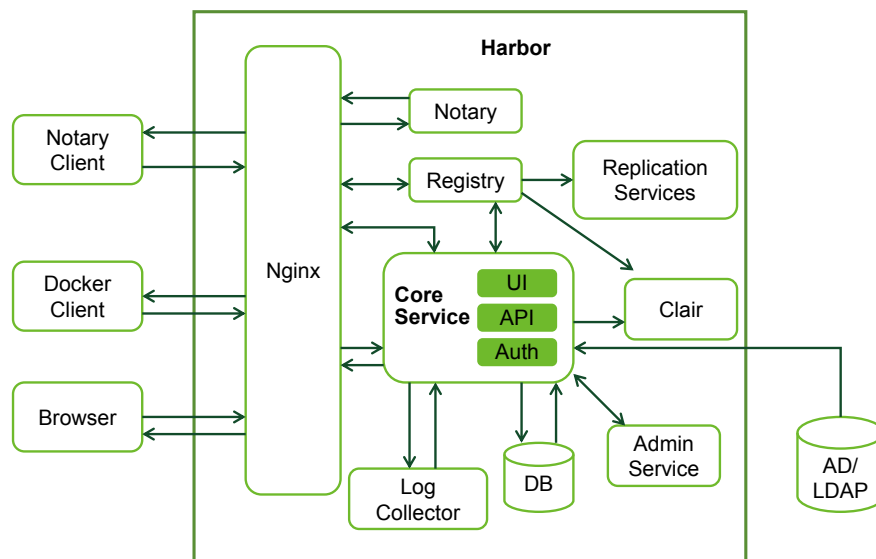
另外，用户也可以将自己创建好的镜像上传至公共仓库供其它的用户使用，软件厂商也可以以镜像形式发布自己的软件。

(2) 私有仓库

不是所有的镜像都适合在互联网进行发布共享，有时候有些含有敏感信息的镜像只能在团队内部共享使用。

因此，考虑到镜像的敏感性与网络的可用性和稳定性等因素，出现了私有仓库的使用场景，或者可以称之为本地仓库。私有仓库是在一定范围内可访问的镜像仓库，典型的私有仓库实现代表为 Harbor^[18]。

图 2.2 Harbor 架构



Harbor 由 VMware 中国研发团队负责开发，旨在帮助用户迅速搭建一个企业级的 Registry 服务。它以 Docker Registry 为基础，提供了管理 UI、基于角色的访问控制、AD/LDAP 集成、审计日志以及镜像漏洞扫描等功能。

表 2.1 Harbor 服务组成

Registry	负责储存 Docker 镜像，并处理 docker push/pull 命令
UI	提供图形化界面，帮助用户管理 Registry 上的镜像，并对用户进行授权
Job Service	Harbor 的任务管理服务
DB	提供数据库服务。负责储存用户权限、审计日志、镜像分组信息等数据
AD/LDAP	提供统一的用户身份认证和权限控制
Log Collector	负责收集其它组件的日志，供日后进行分析
Notary	镜像内容审计，确保镜像真实性，可以选择是否集成
Clair	负责进行镜像的脆弱性扫描，可以选择是否集成

2.1.5 镜像的使用

除了 *docker pull/push* 等常用的命令外，Docker 中操作镜像的一些其它常用操作命令如下，更多镜像操作，可参考 Docker 官方文档^[19]。

(1) docker export

将容器导出到本地。

```
# docker export redis-server -o redis.tar
# ls
redis.tar
```

(2) docker import

将容器从本地以镜像的方式导入。

```
# docker import -m import-test-redis redis.tar
sha256:9cc818fc6b5d62f6d05fa0873885bb4b2cefa20aac5de0fb3cc7907086d166b7
# docker images | grep 9cc
<none>          <none>          9cc818fc6b5d      30 seconds ago    244MB
```

(3) docker save

将 (一个或多个) 镜像导出至本地。

```
# docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
nginx         latest   3c5a05123222   4 days ago    109MB
redis         latest   71a81cb279e3   2 weeks ago    83.4MB

# docker save nginx redis -o test.tar
# ls
test.tar
```

2.2 容器存储

2.2.1 镜像元数据

在 Linux 系统中 Docker 的数据默认存放在 `/var/lib/docker` 中，基于不同的系统又有不同的存储驱动、不同的目录结构。本文以 OCI 标准格式来了解镜像存储的内容。

图 2.3 镜像存储目录

```

.
├── blobs
│   └── sha256
│       ├── 014246127c672bac4a8790acc1acba4fa356fd15b575d660c975acdd46e753de
│       ├── 01729050d692e70a9c961d7caeaa471aebb90185b574a9998d8b7292fce0de1d
│       ├── 3d77ce4481b119f00e53bee9b4a443469c42c224db954ddaa2e6b74cd73cd5d0
│       ├── 73674f4d94033a4285dde32367bb36d6287a12de1c1d272103c7d88f8feaa0e4
│       ├── 7cd2e04cf570947b4db6b63492d7a25772272107153e1cfe328abf5699d17d6b
│       ├── a4903a9d1f01852ddf36bb867aabdcc2ecdc8b28ca2db6f683c823e2088bef10
│       ├── ce7b0dda0c9f3a31a1965c920075d7dc128e6ed444f5a07e1d9bec2a2080625c
│       ├── d266646f40bd8883dddb289a6379a9f38129afa5d9a205dafa464cbf3005847
│       └── e6cde63f2a108dbe06822548128f78d4240f98d26648c fb2172d18d0c35e287e
├── index.json
├── manifest.json
└── oci-layout

```

2 directories, 12 files

镜像每一层的 ID 是该文件内容的哈希校验值，作为该层的唯一标识。获取镜像后，会使用以下方式索引镜像：首先读取镜像的 manifests，根据 manifests 文件中 config 的 sha256 码，得到镜像 config 文件，遍历 manifests 里面的所有 layer，根据其 sha256 码在本地查找，拼出完整的镜像。

2.2.2 存储驱动

理想情况下，使用挂载卷来存储高读写的目录，很少将数据直接写入容器的可写层。但是，总有些特殊需求需要直接写入容器的可写层。这时候就需要存储驱动来作为容器和宿主机之间的媒介。Docker 依靠驱动技术来管理镜像和运行它们的容器间的存储和交互。

目前，Docker 支持 AUFS、Btrfs、Device Mapper、OverlayFS、ZFS 五种存储驱动^[20]。没有单一的存储驱动适合所有的应用场景，要根据不同的场景选择合适的存储驱动，才能有效提高 Docker 的性能。

表 2.2 常见存储驱动对比

	特点	优点	缺点	适用场景
AUFS	联合文件系统未并入内核主线文件级存储	Docker 的第一个存储驱动，比较稳定，且在大量的生产环境中实践过，有较强的社区支持	有多层，写时复制操作时，如果文件比较大且存在比较低的层，可能会慢一些	大并发、少 I/O 的场景
OverlayFS	联合文件系统并入内核主线文件级存储	只有两层	不管修改的内容大小，都会复制整个文件。对大文件进行修改要比小文件修改消耗更多的时间	大并发、少 I/O 的场景
Device Mapper	并入内核主线块级存储	无论是大文件还是小文件，修改时都复制需要修改的块，并不是整个文件	不支持共享存储，当有多个容器读同一个文件时，需要生成多个副本。在很多容器启停的情况下可能会导致硬盘溢出	适合 I/O 密集的场景
Btrfs	并入内核主线文件级存储	直接操作底层设备，支持动态添加设备	不支持共享存储，标识当有多个容器读同一个文件时，需要生成多个副本	不适合在高容器密度的 PaaS 平台上使用
ZFS	把所有设备集中到一个存储池中进行管理	支持多个容器共享一个缓存块，适合大内存的环境	写时复制使碎片化问题更加严重，文件在硬盘上的物理地址会变得不再连续，顺序读会变得性能比较差	适合 PaaS 和高密度的场景

2.2.3 数据卷

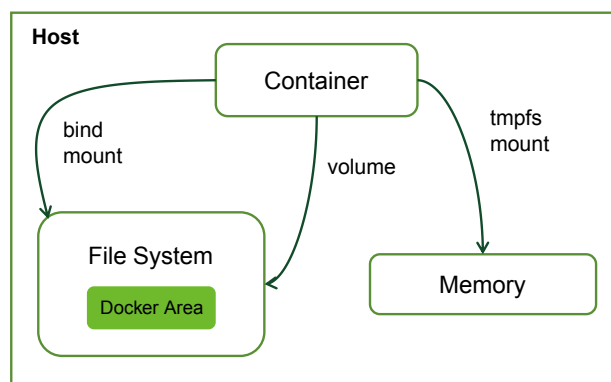
通常，有状态的容器都有数据持久化存储的需求。前一节提到过，文件系统的改动都是发生在最上面的可读写层。在容器的生命周期内，它是持续的，包括容器被停止后。但是，当容器被删除后，该数据层也随之被删除了。

因此，Docker 采用数据卷（Volume）的形式向容器提供持久化存储。数据卷是 Docker 容器数据持久化存储的首选机制。绑定挂载（Bind Mounts）依赖于主机的目录结构，但数据卷是由 Docker 管理。与绑定挂载相比，数据卷有以下几个优点：

- 与绑定挂载相比，数据卷更容易备份或迁移；
- 可以使用 Docker CLI 命令或 Docker API 管理数据卷；
- 数据卷在 Linux 和 Windows 上均可使用；
- 数据卷可以在多个容器之间更安全地共享；
- 数据卷驱动程序允许在远程主机或云上存储数据卷、加密卷的内容或添加其它功能；
- 新数据卷的内容可以由容器预填充。

另外，与使用容器的读写层保存数据相比，数据卷通常是更好的选择。因为使用数据卷存储不会增加容器的大小，并且数据卷是持久化的，不会依赖于容器的生命周期。

图 2.4 Docker 主机上数据卷的挂载方式



2.3 容器网络

从云计算系统的发展来看，业界普遍的共识是，计算虚拟化和存储虚拟化已经不断突破和成熟，但网络虚拟化的发展仍相对滞后，成为制约云计算发展的一大瓶颈。网络虚拟化、多租户、混合云等特性均不同程度地给云网络的安全建设提出全新的挑战。

容器技术提供了轻量级虚拟化的能力，使实例资源占用大幅降低，提升了分布式计算系统的性能，但分布式容器系统的网络仍是较为复杂的部分。

本节将针对容器主机网络以及容器集群网络分别进行介绍。

2.3.1 容器网络支撑技术

容器网络虽然有多种形态，但多数使用了若干种支撑技术，例如网络命名空间（Network Namespace）、Linux 网桥（Linux Bridge）以及虚拟网络接口对（Veth Pair）。

2.3.1.1 网络命名空间

网络命名空间是一种实现网络隔离的技术，创建一个网络命名空间后就有了一个包括网络接口、路由、访问控制规则（Iptables）等网络资源的独立网络环境，该命名空间的网络与其它网络隔离。

2.3.1.2 Linux 网桥

Linux 网桥是 Linux 系统中的虚拟网桥，它可以将不同主机的网络接口连接，从而实现主机间的通信。

Docker 启动后，会默认创建名为 `docker0` 的 Linux 网桥。在未创建任何容器时，可以看到 `docker0` 网桥没有接口连接。

```
# brctl show
bridge name    bridge id        STP enabled    interfaces
docker0        8000.0242d1837f60  no
```

当创建容器时 (*d458f9bd528*)，Docker 会为容器创建虚拟网络接口，并将其连接到 *docker0* 网桥上。

```
# brctl show
```

bridge name	bridge id	STP enabled	interfaces
docker0	8000.0242f22b2de4	no	veth4d91464 veth6ed0a8c

2.3.1.3 虚拟网络接口对

为了实现容器与宿主机网络、外部网络之间的通信，需要通过虚拟网络接口对将容器与 Linux 网桥连接。

当 Docker 启动一个容器时 (*d458f9bd528*)，会创建一个虚拟网络接口对，即两个相连的虚拟网络接口。其中一个连接容器，成为容器 *d458f9bd528* 的网卡 *eth0*；另一个被连接到 *docker0* 网桥上，从而容器内部的数据包先后经过 *eth0* 和 *veth6ed0a8c* 对到达 *docker0* 网桥，实现在同一子网内不同容器之间的通信。

通过下面命令可以得到容器的网卡 *eth0* 在宿主机上的编号 (peer-ifindex)。

```
# docker exec d458f9bd528 ethtool -S eth0
NIC statistics:
peer_ifindex: 37
```

继续执行下面命令找到宿主机上 peer_ifindex 为 37 的接口名称。

```
# ip link | grep 37
37: vethfaa2a17@if36: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master
docker0 state UP mode DEFAULT
```

运行 *ethtool* 可查找其对端接口编号。

```
# ethtool -S vethfaa2a17
NIC statistics:
peer_ifindex: 36
```

2.3.2 主机网络

以 Docker 为例，目前 Docker 容器主机网络 (Host Network) 主要分为以下 4 种模式。

2.3.2.1 None 网络模式

None 网络模式下，容器拥有自己的网络命名空间，但是并不为容器进行任何的配置。创建的容器只有 *loopback* 接口，需要用户为容器添加网卡、配置 IP 等。

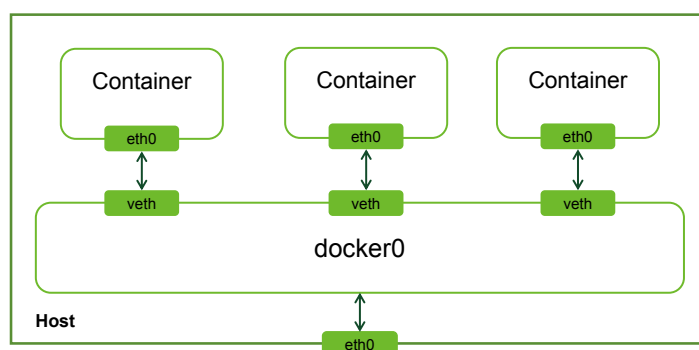
rkt 同样支持 None 网络模式。该网络模式的用途主要是测试容器、稍后为容器分配网络、以及一些对安全性要求高且不需要联网的场景。

2.3.2.2 Bridge 网络模式

Bridge (网桥) 网络模式主要是利用 iptables 进行 NAT 和端口映射，从而提供单主机网络。与虚拟机下的 NAT 网络类似，这种网络模式下同一主机上的容器之间是可以互相通信的，但是分配给每个容器的 IP 地址从主机外部不能访问。

Bridge 网络是 Docker 默认的网络类型，在安装完 Docker 后会默认创建 `docker0` 网桥，并通过虚拟网络接口对连接容器和 `docker0` 网桥，这样主机上的所有容器就处在了一个二层网络中。

图 2.5 Bridge 网络模式



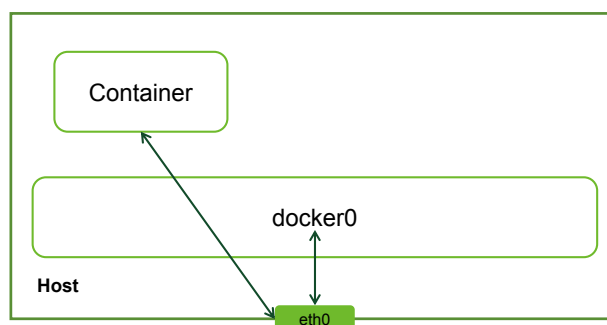
2.3.2.3 Host 网络模式

Host（主机）网络模式下，Docker 服务启动容器时并不会为容器创建一个隔离的网络环境，容器将会被加入主机所在网络，共享主机的网络命名空间（`/var/run/docker/netns/default`）。其网络配置（网络地址、路由表和 `Iptables` 等）和主机保持一致，容器通过主机的网卡和 IP，实现与外部的通信。

由于容器并没有独立的网络命名空间，容器服务的端口没有经过端口映射就直接暴露在主机上，因此容器中服务的端口号不能与主机上已经使用的端口号冲突。

以这种网络模式创建的容器虽然可以访问主机的所有网络接口，但除非在特权模式下部署，否则容器可能不会重新配置主机的网络堆栈。Host 网络模式是 Apache Mesos 中默认使用的网络模式，如果没有指定网络类型，新的网络命名空间将不会与容器相关联，而是与主机网络相关联。

图 2.6 Host 网络模式

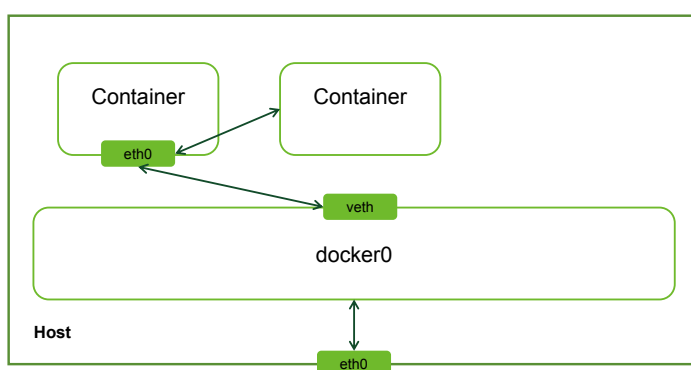


2.3.2.4 Container 网络模式

Container（容器）网络模式比较特殊，新创建的容器和已经存在的某个容器共享同一个命名空间。该新容器不会创建自己的网卡或配置自己的 IP，而是和一个指定容器共享 IP、端口范围等。

这两个容器只有网络方面共享数据，文件系统、进程列表等其它方面还是隔离的。两个容器的进程可以通过 *loopback* 网卡通信。

图 2.7 Container 网络模式



2.3.3 集群网络

本节以 Docker Swarm 和 Kubernetes 为例，具体阐述容器集群中的网络实现。

2.3.3.1 Docker Swarm

Docker 1.19 及以后的版本增加了对 Overlay 网络的原生支持（Docker Swarm），主要是利用网络隧道在主机之间通信。这将允许处于同一 Overlay 网络的不同主机中的容器实现跨主机通信，而同一主机处于不同 Overlay 网络的容器间是不能通信的。这个概念跟云计算中通常认知的 Overlay 网络基本是一致的。Swarm 的 Overlay 网络采用 VXLAN 实现，要求 Linux Kernel 版本不低于 3.19，主要包括如下网络。

(1) Docker_gwbridge 网络

Docker_gwbridge 是在初始化 Docker Swarm 集群后产生的桥接网络，存在于集群中的每个主机节点上，通过它来实现容器和容器所运行的主机间的通信。可以通过下列命令查看 Docker_gwbridge 网络。

```
# docker network inspect docker_gwbridge
.....
"Options": {
  "com.docker.network.bridge.enable_icc": "false",
  "com.docker.network.bridge.enable_ip_masquerade": "true",
  "com.docker.network.bridge.name": "docker_gwbridge"
},
.....
```

由上 *--icc* 配置项为 *false* 可知，连接到 *docker_gwbridge* 网桥的容器之间通过 *docker_gwbridge* 上的接口时不可以通信。

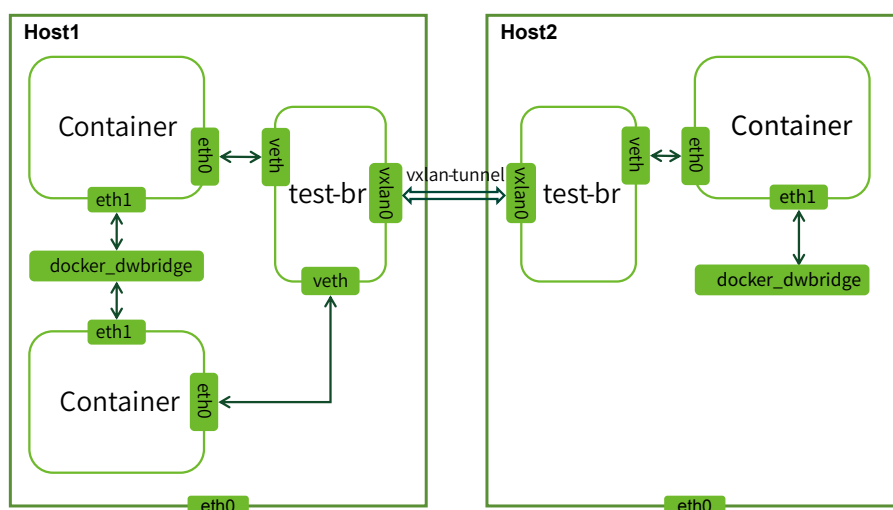
(2) Ingress 网络

Ingress 网络是在初始化 Docker Swarm 集群后产生的 Overlay 网络，其存在于集群中的每一个主机节点上，主要用来实现服务的对外暴露以及 Routing Mesh。有关 Ingress 网络如何实现服务的对外暴露详见 4.5.1 节。

(3) 自定义 Overlay 网络

此网络是用户在初始化 Docker Swarm 集群后，创建服务之前自定义的 Overlay 网络，主要用来实现处于同一 Overlay 网络内的容器间通信。

图 2.8 同一服务的容器网络模型图



上图的三个 Container 是属于同一服务的容器，在 Docker Swarm 模式下同一服务的容器共享用户自定义的 Overlay 网络。图中 test 网络就是自定义的 Overlay 网络，虽然各节点主机上的容器都会连接到 *docker_gwbridge* 上，但其上的接口并不能实现同一主机不同容器间的通信，处于同一服务下容器之间的通信是通过 *test overlay* 网络实现的。

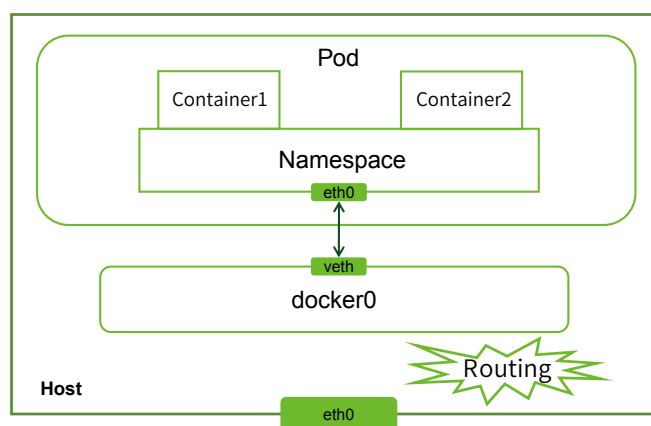
2.3.3.2 Kubernetes

Kubernetes 设计了 Pod 对象，对应某特定应用中的逻辑主机（Logical Host），将每个服务按任务拆分，分别将相应进程包装到相应的 Pod 中。一个 Pod 中包含一个或多个相关的容器，这些容器都会运行在同一个主机中，并且共享相同的网络命名空间和相同的 Linux 协议栈。

一个 Kubernetes 集群通常会涉及到以下三种通信：

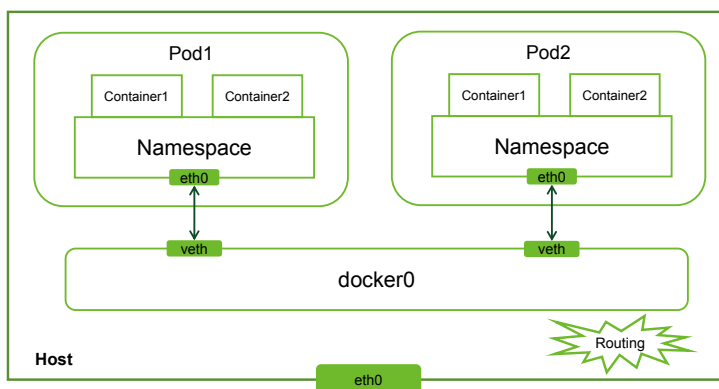
- 同一个 Pod 内，容器和容器之间的通信；
- 同一个主机内不同 Pod 之间的通信；
- 跨主机 Pod 之间通信。

图 2.9 同一 Pod 内容器网络模型



同一个 Pod 内容器之间的通信，由于其共享网络命名空间、共享 Linux 协议栈，因此它们之间的通信是最简单的，这些容器好像是运行在同一台机器上，直接使用 Linux 本地的 IPC 进行通信，它们之间的互相访问只需要使用 localhost 加端口号就可以。

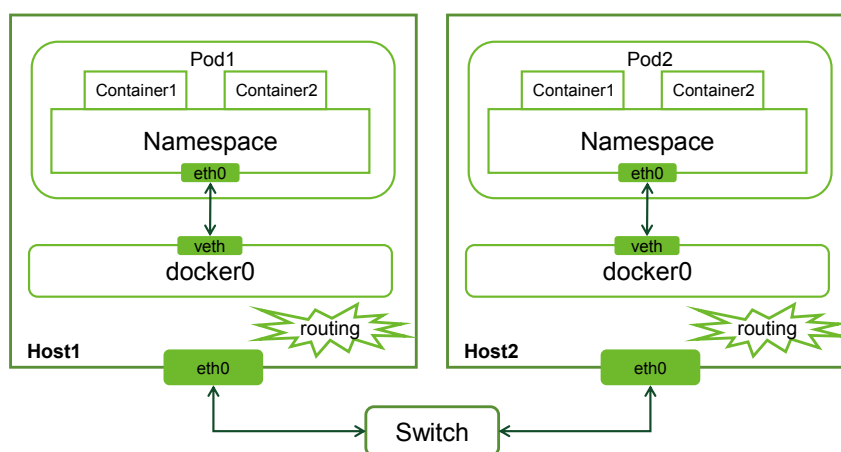
图 2.10 同一主机不同 Pod 间的容器网络模型



同一个主机上不同的 Pod 通过 *veth* 连接在同一个 *docker0* 网桥上，每个 Pod 从 *docker0* 动态获取 IP 地址，

该 IP 地址和 *docker0* 的 IP 地址是处于同一网段的。这些 pod 的默认路由都是 *docker0* 的 IP 地址，所有非本地的网络数据都会默认送到 *docker0* 网桥上，由 *docker0* 网桥直接转发，相当于一个本地的二层网络。

图 2.11 跨主机不同 Pod 间的网络模型

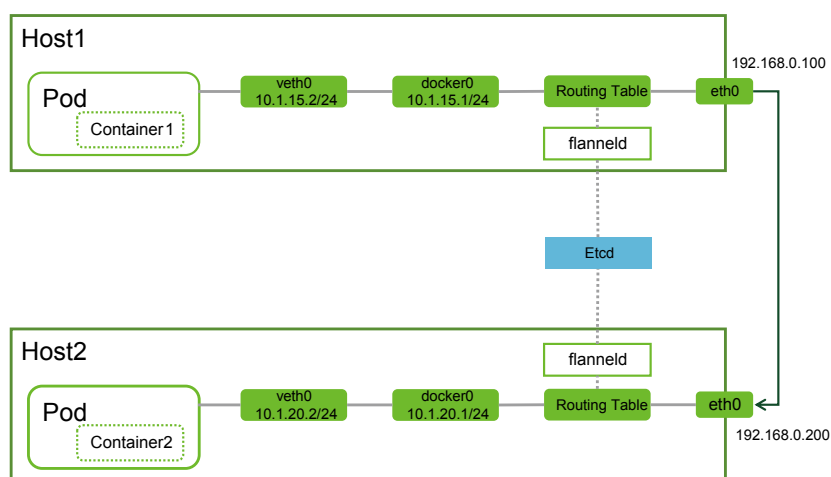


跨主机的 Pod 之间通信较复杂，每个 Pod 的地址和其所所在主机的 *docker0* 在同一个网段，而 *docker0* 和主机的物理网络是不同的网段，如何保证 Pod 内的网络数据能够通过物理网络，找到对端 Pod 所在物理主机地址，并且完成数据传输是通信成功的关键。

因此在 Kubernetes 的网络中，需要有一个全局网络地址规划的模块，当 Pod1 向 Pod2 发送数据时，能够知道目的 Pod2 所在主机的 IP 地址。这样数据从 Pod1 发出，经 Host1 的 *docker0* 网桥路由到 Host1 的物理网卡 *eth0*，然后通过物理网络到达 Host2 的物理网卡 *eth0*、*docker0* 网桥，进而送达 Pod2。

在多数的私有容器云环境中还可以借助第三方开源的网络插件来实现集群网络，比如 Flannel^[21]、Calico^[22] 等。例如，Flannel 是 CoreOS 团队针对 Kubernetes 设计的一个 Overlay 网络工具，其目的在于帮助每一个使用 Kubernetes 的主机拥有一个完整的子网。

图 2.12 Kubernetes Flannel 网络架构



2.4 容器管理与应用

容器技术之所以在微服务、云计算等领域得到广泛应用，除了其轻量化的技术外，弹性、敏捷的容器管理和编排系统支持也是重要因素之一。

2.4.1 容器管理

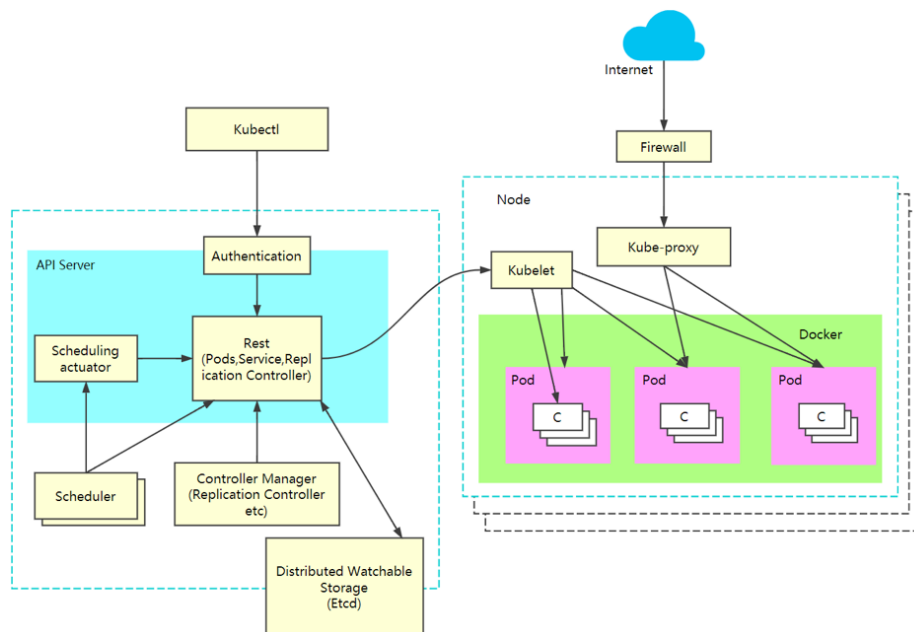
集群化、弹性和敏捷化是容器应用的显著特点，如何有效地对容器集群进行管理，是容器技术落地应用的一个重要方面。集群管理工具（编排工具）能够帮助用户以集群的方式在主机上启动容器，并能够实现相应的网络互联，同时提供负载均衡、可扩展、容错和高可用等保障。当前关注度和使用率比较高的几种容器集群管理工具主要包括：Kubernetes、Apache Mesos、Docker Swarm 和 Docker Compose。

2.4.1.1 Kubernetes

Kubernetes^[23] 是由 Google 基于其内部大规模集群管理系统 Borg 发布的开源分布式容器管理平台，简称 K8S。Kubernetes 为用户提供了集群管理能力、多租户应用支撑能力、透明的服务注册和服务发现机制、负载均衡能力以及故障发现和自我修复等能力。

Kubernetes 的架构如下图所示，主要由一个或多个 Master 节点和一个或多个 Node 节点组成。

图 2.13 Kubernetes 架构



Master 节点包含多个重要组件，例如和存储相关的 Etcd 组件、和调度相关的 Scheduler 组件、集群控制高可用组件 Controller Manager、与外部通讯协调整个集群的组件 API Server 等。

Node 节点含有两个重要的组件，分别是管理集群中 Pod 生命周期的组件 Kubelet 和负责 Kubernetes 中网络配置的组件 Kube-proxy。

在 Kubernetes 中，包括以下几个重要操作单元：

- **Pod:** Kubernetes 中运行应用或服务的最小单元，其设计理念是支持多个容器在一个 Pod 中共享网络地址和文件系统。
- **Service:** 访问 Pod 的代理抽象服务，主要用于集群内部的服务发现和负载均衡。
- **Replication Set:** 用于伸缩 Pod 副本数量的组件。
- **Scheduler:** 集群中资源对象的调度控制器。
- **Controller Manager:** 负责集群中资源对象管理同步的组件。
- **Etcd:** 分布式键值对 (k, v) 存储服务，存储整个集群的状态信息。
- **Kubelet:** 负责维护 Pod 的生命周期。
- **Labels:** 一组键值对，与某种资源关联以提供标识定位能力的属性。
- **Deployment:** 用于管理 Pod 对象，集成了上线部署、滚动升级、创建副本、暂停 / 恢复上线任务、回滚等功能。
- **Volumes:** 即目录，其中存有数据，在容器启动时指定参数会自动挂载至容器内部，Volumes 分静态和动态，其具有不同的生命周期。
- **Stateful Set:** 通常创建的 Pod 是无状态的，这样会导致一旦 Pod 挂掉再重启后会找不到之前挂载的 Volume，所以可以通过 Stateful Set 来保留 Pod 的状态。

Kubernetes Pod 工作流程为：

(1) 提交请求

用户通常提交一个 yaml 文件，向 API Server 发送请求创建一个 Pod，yaml 文件含有此 Pod 的详细信息，包含此 Pod 运行副本数、镜像、Labels、名称，端口暴露情况等。API Server 接收到请求后将 yaml 文件中的 spec 数据存入 Etcd 中。

(2) 资源分配

Scheduler 通过 API Server 的 watch 接口定时监听 Etcd 数据库中资源的变化（此处指上一步待分配的 Pod），当监测到了 Pod 后，通过 Scheduler 的调度策略选择出具有运行 Pod 能力的 Node 节点并将 Pod 与目标 Node 节点进行绑定，同时更新 Etcd 数据库中 Pod 的分配情况。

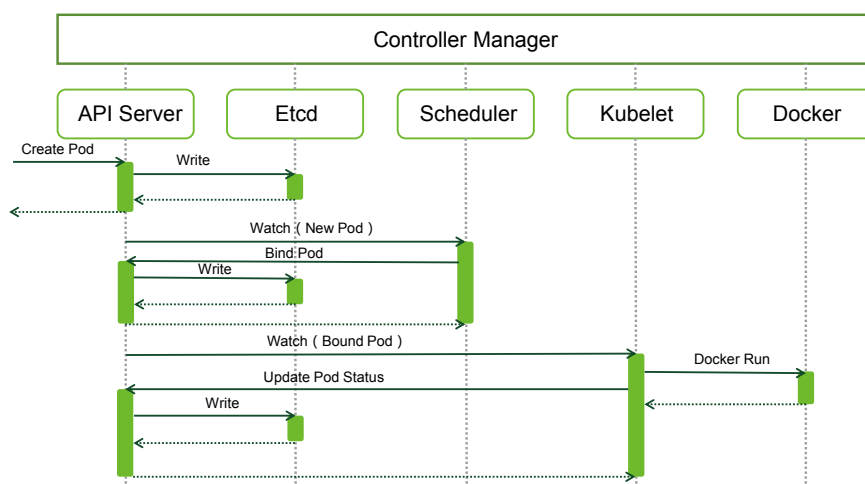
(3) 新建容器

此时目标 Node 节点上的 Kubelet 通过 API Server 的 Watch 接口监测到 Etcd 中 Pod 的分配信息，同时将 Pod 的相关数据传递给容器运行时以负责此 Pod 的整个生命周期，之后 Kubelet 还会通过容器运行时获取 Pod 的状态信息并通过 API Server 更新至 Etcd。

(4) 资源状态同步

为了保证此 Pod 在 Node 节点中运行正常（Pod 可能会因为某些原因被杀死），Controller Manager 中的 Replication Set 组件通过 API Server 定时监听 Etcd 以获得 Pod 的最新状态并最终对 Pod 进行数量上的同步，从而保证了 Pod 运行副本数与用户指定副本数相同。

图 2.14 Kubernetes 创建 Pod 流程示意图

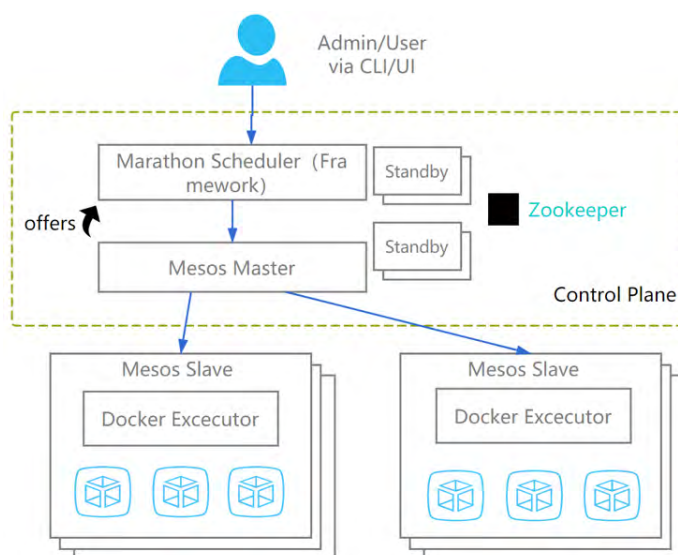


2.4.1.2 Apache Mesos

Apache Mesos^[24] 源自 UC Berkeley 的一个对集群资源进行抽象和管理的开源项目，后在 Twitter 得到广泛使用。Apache Mesos 将整个数据中心的资源进行抽象和管理调度，有类似主机操作系统的功能，使得多个应用同时运行在集群中分享资源。

Apache Mesos 的架构如下图所示，由一个或多个 Master 节点和一个或多个 Slave 节点，以及若干个 Framework 应用程序组成。

图 2.15 Apache Mesos 架构



Master 主要负责管理各个 Framework 和 Slave 节点，并将 Slave 节点上的资源分配给相关的 Framework。Slave 负责管理当前节点上的任务分配。Framework 是外部计算框架，可插拔，常用的有 Marathon、Hadoop、MPI 等，Framework 可通过注册的方式接入 Mesos，以便 Mesos 进行统一管理和资源分配。

Framework 主要由两个组件组成，分别是调度器和执行器。调度器与 Master 交互，根据可用资源，将任务调度至 Slave 中加载；执行器从 Framework 中获取变量，在 Slave 节点中运行任务。

Apache Mesos 工作流程主要为：

(1) 资源信息聚合

Slave 节点向 Master 节点上传其可用资源信息，Master 节点作为一个资源池对收集到的资源进行维护。

(2) 任务分配

Master 节点将可用资源提供给某一应用的 Framework Scheduler，该应用的 Framework Scheduler 对比提供的资源是否满足其任务的运行要求，如果不满足，则向 Master 节点传递拒绝提供的资源信息，而 Master 节点则再次向其它 Framework 提供资源；如果满足，Framework Scheduler 则接受提供的资源信息。

(3) 任务下发

Framework Scheduler 下发任务至 Master 节点，Master 节点接收并在 Slave 节点中使用 Executor 运行应用。

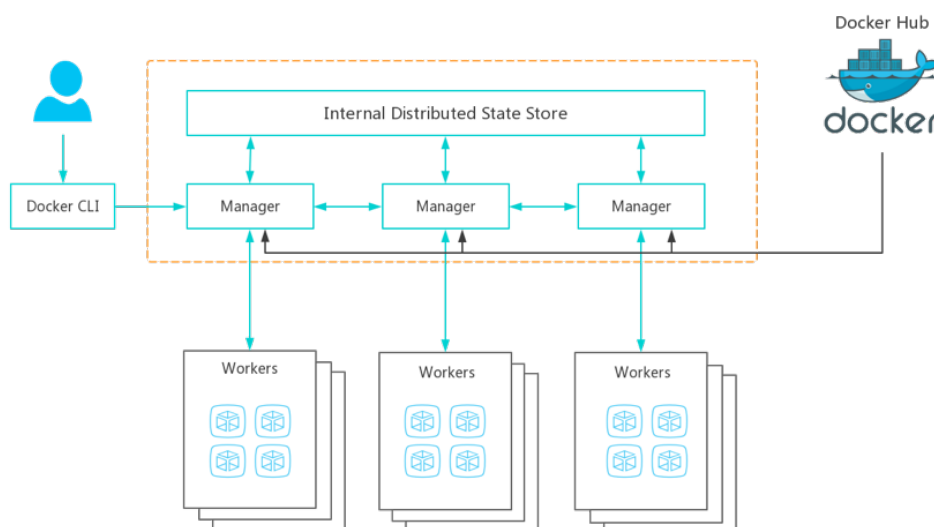
2.4.1.3 Docker Swarm

Docker Swarm^[25] 是 Docker 公司在 2014 年 12 月初发布的一款容器管理工具，其功能较为简单，主要用于管理 Docker 集群，使得 Docker 集群以一个虚拟整体暴露给用户。

作为 Docker 的一个组件，Docker Swarm 最大的优势是其 API 集成到了标准的 Docker API，这使得 Docker 应用开发者可以很容易与 Docker Swarm 进行集成。目前，包括 Rackspace 等平台都采用了 Docker Swarm，用户也很容易在 AWS 等公有云平台使用 Docker Swarm。

Docker Swarm 由若干管理节点 Manager 和工作节点 Worker 组成，架构如图所示。Manager 节点负责接收来自用户的请求，并将任务分配至 Worker 节点，Manager 节点同时也可以执行 Worker 节点的工作职责。Worker 节点接收来自于 Manager 节点调度的任务并运行。

图 2.16 Docker Swarm 架构



所有的节点都具有基本的 Docker 运行组件：Docker Daemon、负载均衡，为容器的运行提供了环境。以下介绍几个重要组件：

- **Master 节点：**主要由 Manager CLI（Swarm 命令行工具）、Scheduler（调度器）、Discovery Service（服务发现）组件构成；
- **Slave 节点：**主要组件为 Swarm Agent；
- **服务发现组件：**集群中的物理和通讯环境复杂，服务中断无法准确预测，服务发现组件为集群提供了服务保障。当某个 Slave 节点宕机之后，服务发现组件会快速发现，通过调度器快速在另一个节点恢复已经失效的服务。

Docker Swarm 工作流程为：

（1）集群操作

用户使用 Master 节点的 Manage CLI 对应用部署进行规划。

（2）应用分发

用户使用 Manage CLI 对应用部署规划之后，由调度器对应用进行集群分发，Swarm Agent 负责节点间通讯的传输。

（3）服务高可用

Docker Swarm 使用其服务发现组件维护 Slave 节点中运行的应用信息，保持集群中稳定运行应用的数量，以达到集群应用高可用的目的。

2.4.1.4 Docker Compose

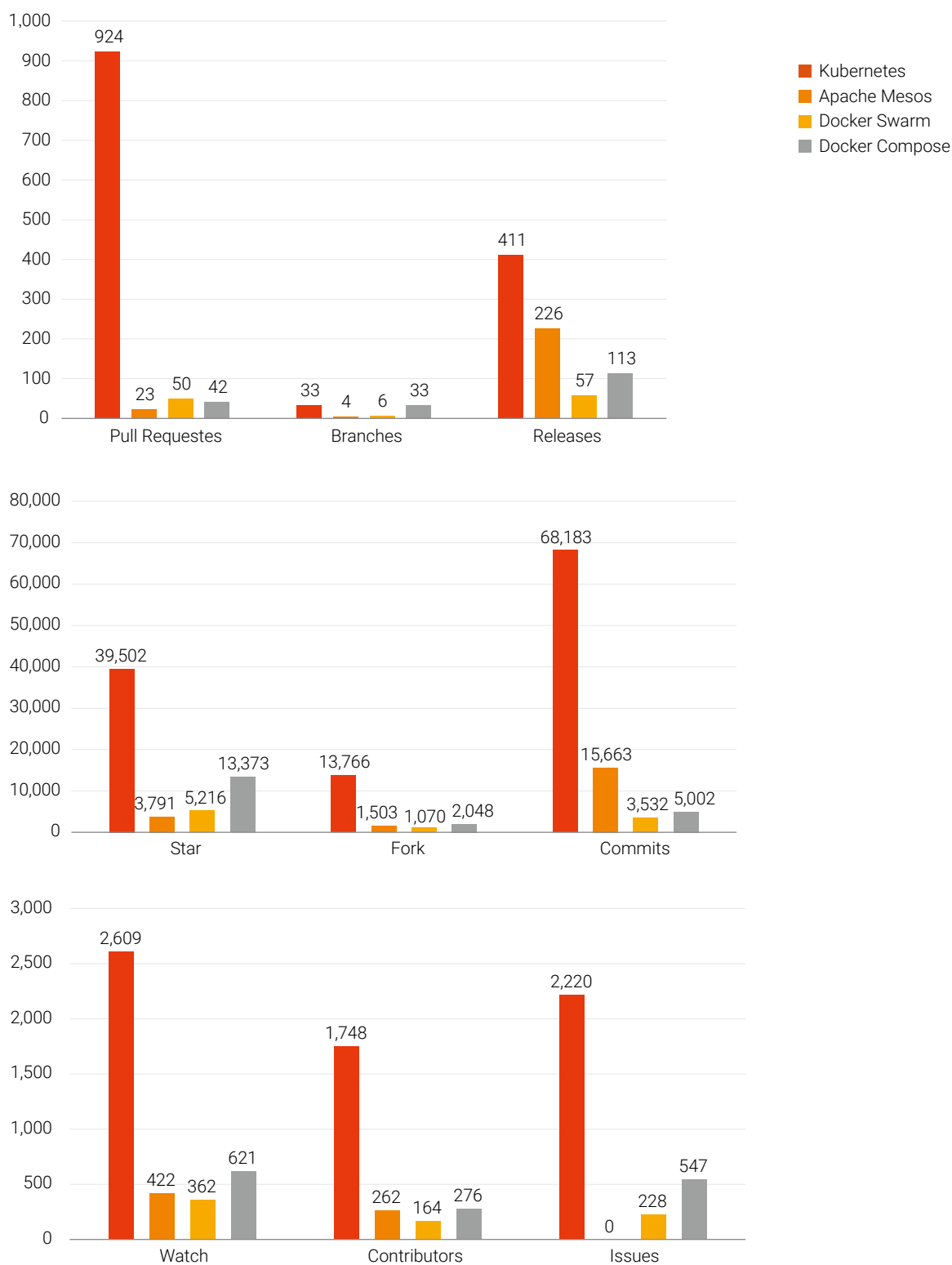
Docker Compose^[26] 是 Docker 推出的另一个编排工具，用于对用户需要的应用服务进行描述和部署。需要说明的是，Docker Compose 只能在单节点上进行应用的部署，如果编排场景是设定在集群中，那么还需要依赖 Docker Swarm 来进行应用的集群化部署。

2.4.1.5 工具对比

本节对 Github 上 Kubernetes、Apache Mesos、Docker Swarm、Docker Compose 项目的活跃程度进行了对比分析，分析结果从一定程度体现了开源社区对这四款编排工具的关注和支持程度。

从 Pull Requests、Issues、Commits、Branches、Releases、Contributors、Watch、Star、Fork 等指标可以看出，开发者们对 Kubernetes 的关注度最高，其后依次是 Apache Mesos、Docker Swarm 与 Docker Compose。

对比 Releases 可以发现，Kubernetes 与 Apache Mesos 的迭代总量较大，其次是 Docker Compose 与 Docker Swarm。那么将项目关注度和项目迭代总量结合起来看，不难发现 Kubernetes 与 Docker Mesos 目前处于编排工具的第一梯队，Docker Swarm、Docker Compose 热度则较为缓和，处于第二梯队。

图 2.17 Github 容器编排工具活跃度对比¹

¹ 数据统计截至 2018-7-31。

除了上述工具外，Rancher^[27]、Docker Machine^[28] 等也是常见的容器集群管理工具，不同工具的侧重点和功能点不尽相同。正是这些集群管理工具的快速发展，使得容器技术在生产、落地的时候更容易被用户所接受和使用。

2.4.2 容器使用场景

容器提供了类似虚拟机的隔离功能，但系统资源消耗比虚拟机少得多，近年来在多个场景中得到了越来越广泛的应用。

中国信息通信研究院 2018 年发布的《中国云计算开源发展调查报告》^[29] 显示，30.1% 的企业已经将容器技术投入生产环境使用，较 2016 年增长了 6.8%，正在测试环境使用容器的企业占比为 36.3%，还有 24.5% 的企业正在对容器技术进行评估。技术普及度不够（43.7%）、缺少成功案例（41.4%）和对安全性的担忧（33.8%）是很多企业尚未使用容器技术的三个最主要原因。

该调查报告还对容器的使用场景进行了统计，其中 57.9% 的企业将其用于运维自动化，占比最高。其它使用场景还包括开发测试的快速交付（34%）、多环境一致性管理（31.2%）、CI/CD（29.4%）以及微服务（22.7%）。

本文将容器技术的使用场景归纳总结为云计算中的容器服务、DevOps、微服务、运维自动化等。下面本节将就这几个主要使用场景进行简要的介绍和分析。

2.4.2.1 云计算

平台即服务（Platform as a Service, PaaS）和容器即服务（Container as a Service, CaaS）都是云计算的服务模式，是将软件研发的平台或者业务基础平台作为一种服务提供给用户。由于其对 DevOps、CI/CD 天然的优势，CaaS 正如雨后春笋般发展起来。

无论是国外的谷歌、微软或亚马逊，还是国内的华为、阿里、腾讯等云服务商，均在其公有云中提供了容器服务。例如谷歌的 GKE^[30]（Google Kubernetes Engine）、微软的 AKS^[31]（Azure Kubernetes Service）和 ACI^[32]（Azure Container Instances）、亚马逊的 ECS^[33]（Elastic Container Service）和 EKS^[34]（Elastic Container Service for Kubernetes）。

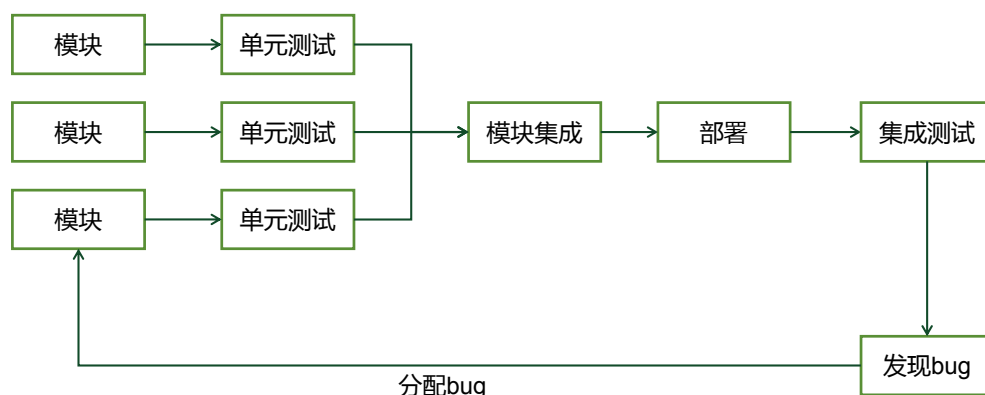
据 RightScale 2017 年的调查报告^[35]，在云服务商提供的服务中，Docker 的使用占据了很大的比重，包括 AWS ECS（35%）、Azure 容器服务（11%）以及谷歌容器引擎（8%）。

国内有包括 DaoCloud、数人云、时速云、灵雀云等众多企业级 PaaS 云服务提供商，对接金融、政府和电信等行业，助力企业打造领先的云原生应用平台。其中绝大多数的服务提供商，在技术上也均选择了 Docker 作为其容器运行时引擎。

2.4.2.2 DevOps

当前软件系统的开发过程通常包括编码、单元测试、集成测试、处理 bug 等步骤，如图 2.18 所示。随着系统复杂度的增加，模块之间的依赖关系越来越复杂，很多 bug 要到项目集成时才能发现，而且距离开发阶段越久，bug 的修复成本就越高。

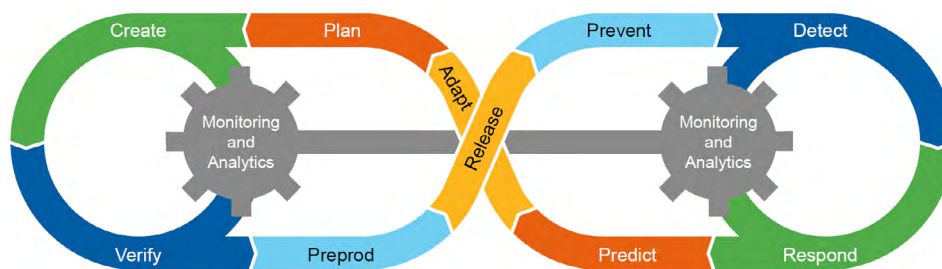
图 2.18 传统软件系统开发过程



DevOps 是一个面向 IT 运维的完整 workflow，以 IT 自动化和持续集成 (CI)、持续部署 (CD) 为基础，来优化开发、测试、运维等所有环节。

DevOps 成为开发和运营的新组合模式，强调软件开发人员和运维人员的沟通合作，通过自动化流程来使得软件构建、测试、发布更加快捷、频繁和可靠。缩短了各个环节的等待时间，减少了很多重复性、手工性的工作，使得解决问题的成本明显降低，成为解决前述问题的有效手段。

图 2.19 DevOps 能力闭环



DevOps 成功的关键标准之一是提升开发对运营的影响，容器技术可在整个 DevOps 闭环中发挥重要作用。在实践中，开发团队不应该只将代码丢给运营团队，还应该关心代码如何在生产环境中运行，于是可利用容器技术将应用程序以容器镜像的形式发布，这些镜像不仅包括应用程序代码，还包括基本操作系统以及依赖库等整个堆栈，甚至还配置好了运行时环境，从而完成了应用一致性地部署，极大降低了运营成本。

RightScale 2017 年的调查报告^[35]显示，超过 84% 的企业，DevOps 的采用率达到 78%，而众多的 DevOps 工具中，Docker 的使用率也达到了 35%，Kubernetes 的使用率也从 2016 年的 7% 增长到 2017 年的 14%。

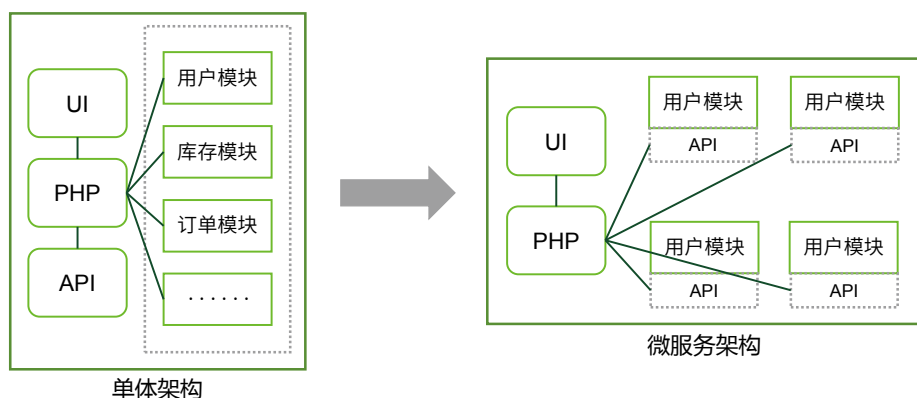
2.4.2.3 微服务

微服务，就是将一个完整应用中所有的模块拆分成多个不同的服务，其中每个服务都可以进行独立部署、维护和扩展，服务之间通常通过 RESTful API 进行通信。

微服务设计的本质在于用功能较明确、业务较精炼的服务去解决更大、更实际的问题。微服务架构相比于传统的单体架构其最大优点在于降低了各模块之间的耦合度。每个服务都可由一个团队来进行开发和维护，不同的服务甚至可以用不同的编程语言、架构来实现，这样每个服务都可用其最适合的技术加以实现，不仅增加了应用开发效率，而且使得整个架构清晰明了，降低了学习成本。

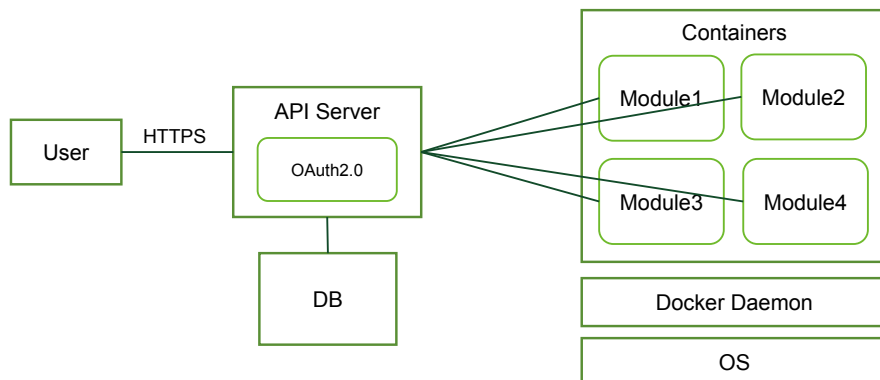
下图是从单体架构到微服务架构转换过程的示例，单体架构将 PHP 服务和模块的功能耦合起来，而微服务架构基于服务概念对应用进行了拆分，为每个适合拆分的模块提供了 API，PHP 服务通过对 API 的逻辑管理达到对外提供服务的目的。

图 2.20 单体架构和微服务架构对比



微服务的核心架构，主要包含两大组件：API Server 以及微服务应用。其中 API Server 是整个微服务的核心，集成了认证、授权、请求日志记录、请求日志监控、负载均衡等功能，微服务应用可用容器进行封装，在系统内部提供 REST API 接口。以下是基于容器的微服务架构示例。

图 2.21 基于容器的微服务架构



随着微服务架构的兴起，业界诞生了许多类型的微服务架构，第一代微服务架构典型的有 Spring Cloud¹、Dubbo² 等，第一代微服务架构因其稳定性等优势，目前使用者较多，已被许多公司用于生产环境中。第二代微服务架构为服务网格（Service Mesh³），目前已衍生出许多种类型的 Service Mesh，典型的有 Buoyant 公司开发的 Linkerd^[36]、Google、IBM、Lyft 协作开发的 Istio^[37] 等。

2.4.2.4 快速部署交付

前述应用场景主要是模式和设计，此外，容器的一个重要特点是可将复杂系统快速部署交付，而省去了大量安装和配置的工作。一个典型的场景是可以采用容器的方式部署 OpenStack 等基础设施平台，实现其运维管理的自动化和敏捷化。

Kolla^[38] 是 OpenStack 中用于将所有组件进行容器化部署的项目。众所周知，对于 OpenStack 的安装和部署，通常要消耗大量的精力，如何做到把更多的精力放在业务逻辑实现，而不是平台环境的安装和部署，是大家一贯的诉求。

使用 Kolla 来部署 OpenStack，装好操作系统之后，大概需要 10 分钟左右的时间，就可以搭建出包括各种社区功能的 OpenStack 系统。由于这种容器化部署，基本是类似于搭积木的方式，所以无论是对于安装，还是升级，都会变得更优雅。

---

1 Spring Cloud 是一个基于 Spring Boot 实现的云应用开发工具，它为基于 JVM 的云应用开发中的配置管理、服务发现、智能路由、微代理等操作提供了一种简单的开发方式。

2 Dubbo 是阿里巴巴公司开源的一个高性能服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入，可以和 Spring 框架无缝集成。

3 Service Mesh 是一个基础设施层，用于处理微服务间通信。在实际应用当中，Service Mesh 通常是由一系列轻量级的网络代理组成的，其与应用程序部署在一起，对应用程序透明。



2018 绿盟科技
容器安全技术报告

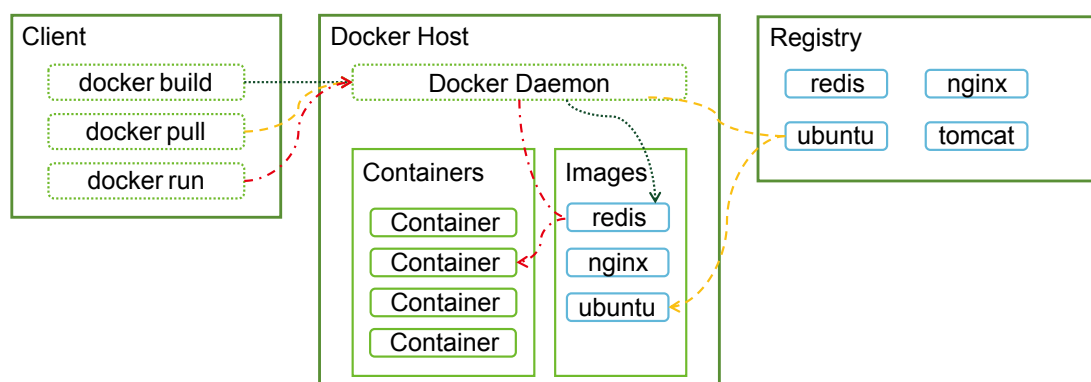
3. 安全风险和挑战

3.1 脆弱性和安全风险分析	33
3.2 安全威胁分析	41
3.3 容器应用安全威胁	43

3.1 脆弱性和安全风险分析

作为容器技术的一种具体实现，Docker 近年来受到越来越多的关注，在某种程度上，Docker 已经成为了容器技术的代表。Docker 在设计上，采用了常见的 Client/Server (C/S) 架构，在 Docker 主机运行 Docker 服务程序 (Docker Daemon)，Docker Client 根据需求向 Docker 服务程序发出相关的请求，其架构如下图所示。本节将以 Docker 为例，介绍容器的脆弱性和安全风险。

图 3.1 Docker C/S 架构示意图



3.1.1 软件风险

作为一款软件，Docker 在软件设计及代码实现上，会有一些安全风险，本小节分别从 Docker 的软件设计和代码漏洞两个层面分析其存在的安全风险。

3.1.1.1 软件设计

Docker 的设计虽然实现了良好的操作系统级隔离，但同时也存在很多安全隐患，比如其默认的组网模式以及与主机共享操作系统内核、共享主机资源、采用 Linux Capabilities 机制、隔离的不充分等，接下来分别对以上几种安全隐患可能导致的攻击作以说明。

(1) 局域网攻击

从网络实现来看，Docker 支持多种组网模式，在默认的桥接模式组网下，即同一主机上创建的容器都桥接挂在网桥 `docker0` 上，这样同一主机上的容器之间可以构成局域网，因此针对局域网的 ARP 欺骗、嗅探、广播风暴等攻击方式将会对这些容器造成安全威胁。所以在一个主机上部署多个容器需要进行合理的网络配置，设置访问控制规则，实现网络之间的隔离和边界。

(2) 拒绝服务攻击

容器实例共享主机的资源，也就是说底层的 CPU、内存和磁盘等资源由主机操作系统进行统一的调度和分配。如果不对每个容器的可用资源进行有效的限制和管理，就会造成容器之间资源使用不均衡，严重时可能导致主机和集群资源耗尽，造成拒绝服务。

(3) 有漏洞的系统调用

Docker 与虚拟机的主要区别是，Docker 与主机共用一个操作系统内核，如果主机内核存在横向越权或者提权漏洞，尽管容器是以非特权模式运行的，但容器中的攻击者还是可以利用内核漏洞逃逸到主机，进行恶意行为操作。

(4) 特权模式共享 root 权限

既然容器与宿主机共用一个操作系统内核，当以特权模式运行容器时，容器内的 root 用户也就拥有了主机的 root 权限，进而可以在容器中进行几乎无限制的操作。

(5) 未隔离的文件系统

虽然 Docker 已经对文件系统进行了隔离，但是有一些重要的系统文件暂时没有被隔离，如 `/sys`、`/proc/sys`、`/proc/bus` 等。

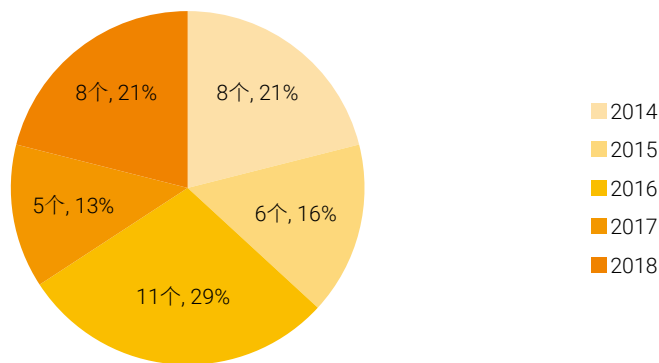
针对以上可能造成的攻击，Docker 已有了对应防护方案，只要根据防护方案做好相应配置即可以避免以上问题，具体可参考官网文档^[39]，此处由于篇幅限制，不再赘述。

3.1.1.2 软件漏洞

根据国家信息安全漏洞库^[40]的统计，截至 2018 年 7 月 31 日，Docker 相关的漏洞共 38 个，其中包括 Cisco、boot2docker、Jenkins 等厂商产品或开源项目。

图 3.2 显示了所有漏洞的发布数量随时间的分布情况，从图中可以看出，每年都会有一定数量的 Docker 漏洞公布出来，说明软件在漏洞上并没有减少的趋势，还是存在一定的安全风险。

图 3.2 历年 Docker 漏洞情况汇总¹



在所有标注了危害等级的 33 个安全漏洞中，中危及以上的漏洞占比达到 90% 以上，高危和超危漏洞占 36%。CVSS 评分 10 分的超危漏洞就包括 4 个，其中 CVE-2014-9357 出现在 Docker 1.3.2 版本中，远程攻击者可借助 LZMA(.xz) 归档的 Dockerfile 中特制的 ‘image’ 或 ‘build’，利用该漏洞以 root 权限执行任意代码；CVE-2016-9223 是 Cisco CloudCenter Orchestrator 中的 Docker Engine 配置安全漏洞，该漏洞源于程序没有正确配置文件，远程攻击者可利用该漏洞提升权限在受影响的系统中部署 Docker 容器。图 3.3 显示了所有漏洞的风险程度汇总。

¹ 数据统计截至 2018-7-31。

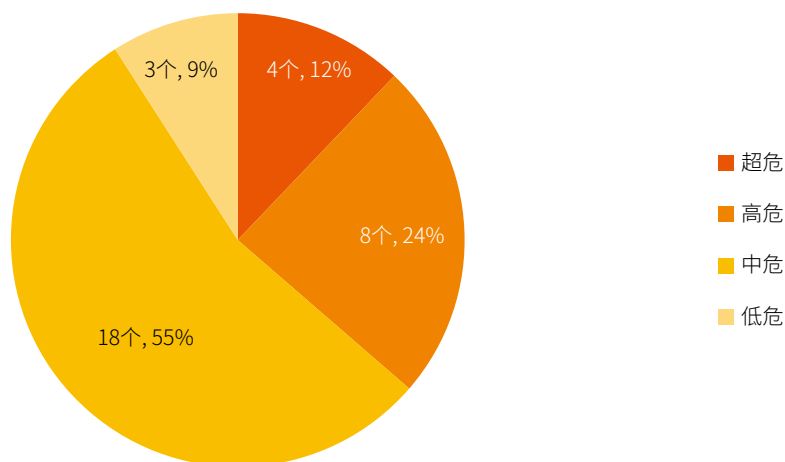
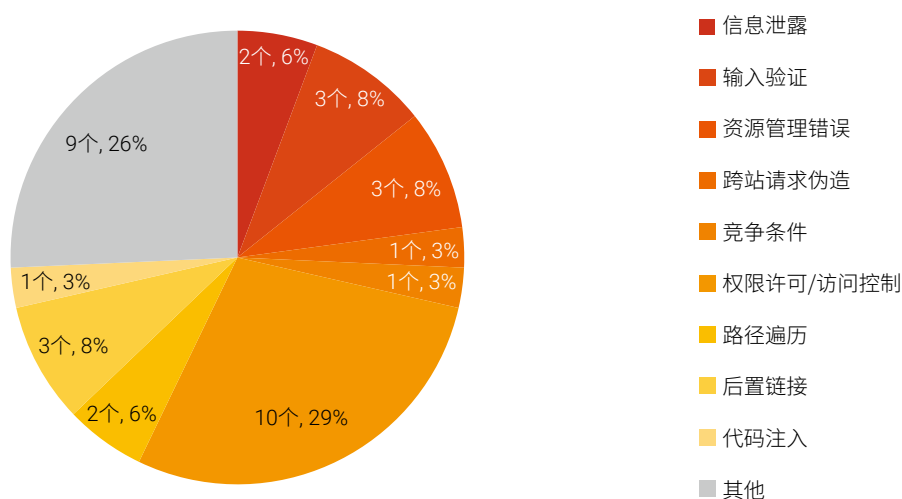
图 3.3 Docker 漏洞风险程度汇总¹

图 3.4 对所有漏洞可能造成的安全风险进行了统计分析，从图中可以看出，Docker 漏洞隐藏的风险主要包括权限和访问控制、后置链接、资源管理错误、输入验证等。比如 Docker Engine 1.12.2 版本中存在的 CVE-2016-8867 漏洞就是权限和访问控制威胁例子，该漏洞源于启用的环境条件配置了错误的条件策略。攻击者可借助恶意的镜像利用该漏洞绕过用户许可，访问文件系统容器或挂载磁盘中的文件。

图 3.4 Docker 漏洞影响汇总²

3.1.2 API 接口安全

按照 Docker 的实现架构，Docker 服务默认监听在 Unix Socket 上，比如 `unix:///var/run/docker.sock`，为了实现集群管理，Docker 官方还提供了一个远程管理接口的 REST API，允许通过 TCP 远程访问 Docker 服务。

¹ 数据统计截至 2018-7-31。

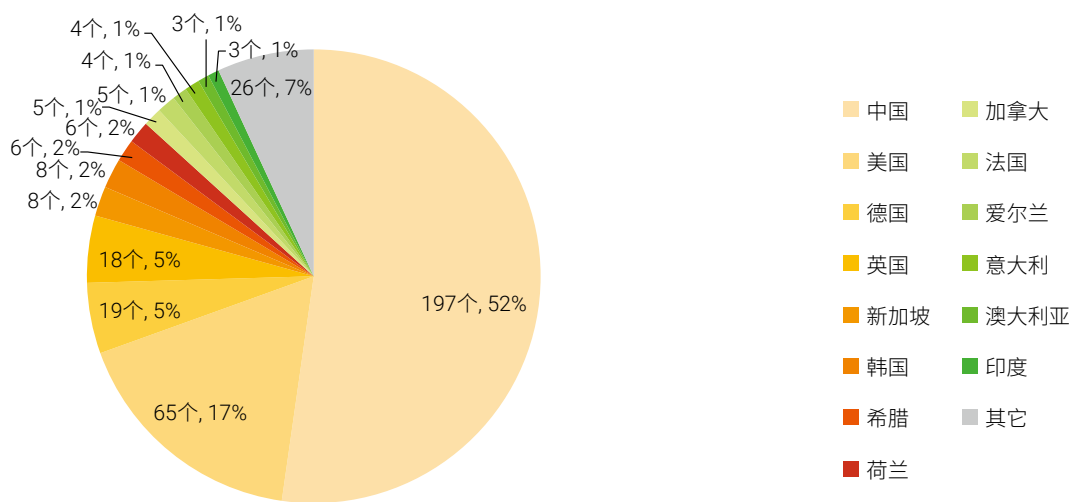
² 数据统计截至 2018-7-31。

例如在使用 Docker Swarm 时，Docker 节点上会开放一个 TCP 端口 2375，绑定在 0.0.0.0 上。Docker 服务作为守护进程运行在后台，这个守护进程默认会采用非加密端口的形式，将服务监听端口 2375 暴露出来，也可以通过执行以下启动参数 `dockerd -H=0.0.0.0:2375 -H unix:///var/run/docker.sock` 让 Docker 监听本地所有地址的 2375 端口，这样本地守护进程就可以执行接收到的 Docker 命令请求。例如，通过执行 `docker -H tcp://$HOST:2375 ps` 这条指令，可以查看到 \$HOST 主机上所有的 Docker 实例。既然能够执行 `docker ps` 命令，那么同样可以执行 `docker run`、`docker rm` 等。

但是开启这种没有任何加密和访问控制的 Docker Remote API 服务是非常危险的。尤其是将默认的 2375 端口暴露到互联网中，一旦被攻击者发现，攻击者无需认证即可访问到容器数据，从而导致敏感信息泄露；也可以恶意删除容器上的数据，或可进一步利用容器自身特性，直接访问主机上的敏感信息，获取服务器 root 权限，对敏感文件进行修改并最终完全控制服务器。

2018 年 5 月 -7 月我们利用绿盟威胁情报中心 (NTI) [41] 对全网的 2375 端口进行检索，发现这段时间暴露在互联网上的 2375 端口地址达 337 个。图 3.5 显示了在全球范围内暴露主机的分布情况，从图中可以看出，主机暴露数据覆盖多达 29 个国家，这个数据一方面说明了 Docker 的受欢迎程度，另一方面也说明了用户对于 Docker 的使用还不是很规范，需要规范操作来规避风险。

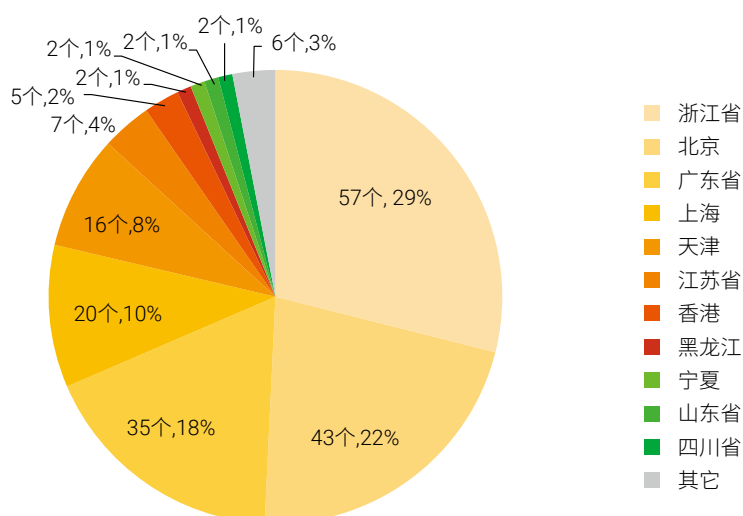
图 3.5 互联网暴露 2375 端口主机全球分布图



针对前述 337 个服务的 IP 地址，本文针对地理区域进行了统计。从图 3.4 中可以看出，在全球范围内，互联网上暴露的 Docker 服务主要分布于中国、美国以及德国，其中中国有 197 个 IP 地址以 52% 位居第一，美国有 65 个 IP 地址以 17% 位居第二，德国有 19 个 IP 以 7% 位居第三。

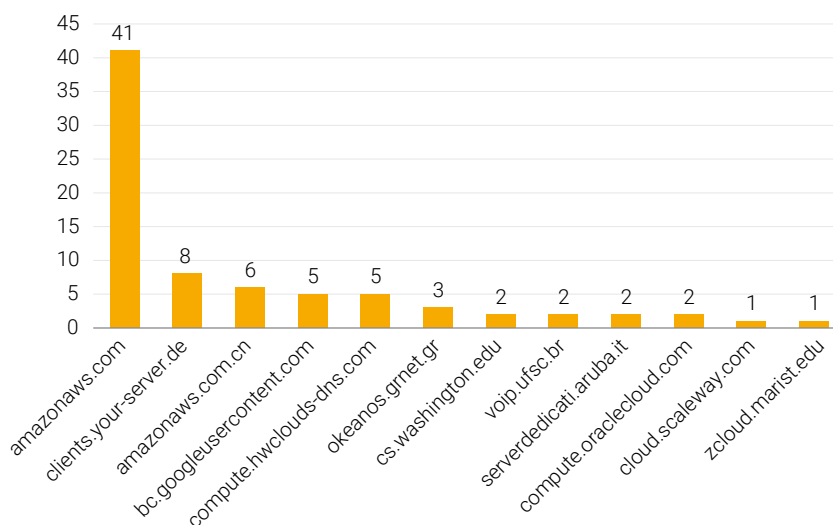
从图 3.6 中可以看出，在中国范围内，互联网上暴露的 Docker 服务主机主要存在浙江、北京以及广东。其中浙江有 57 个 IP 占比 29%，北京有 43 个 IP 占比 22%，广东有 35 个 IP 占比 18%。

图 3.6 2375 端口暴露地址国内分布图



针对前述暴露的 337 个 Docker 服务 IP，我们对其域名服务分布情况进行了统计，其中不乏某些知名公有云厂商的 IP 地址。

图 3.7 暴露地址全网域名分布图



需要说明的是，将 2375 端口暴露在互联网属于错误配置，而非 Docker 的漏洞。关于这个问题，在 Docker 官方使用文档中也有明确的警告：如果将 Docker 服务绑定到一个互联网的 TCP 端口，将可能使得恶意用户获取主机的 root 权限。

Warning: Changing the default docker daemon binding to a TCP port or Unix docker user group will increase your security risks by allowing non-root users to gain root access on the host. Make sure you control access to docker. If you are binding to a TCP port, anyone with access to that port has full Docker access; so it is not advisable on an open network.

除了对 Docker Daemon 远程访问端口 2375 暴露情况进行分析外，我们在 2018 年 7 月也分析了 Kubernetes 的暴露服务情况，利用 NTI 对全网的 6443 端口（Kubernetes 的 API Server 默认 SSL 端口）进行扫描分析，发现这段时间暴露在互联网上的 Kubernetes 服务达 12803 个。图 3.8 显示了在全球范围内的 Kubernetes 服务暴露分布情况。

图 3.8 暴露的 Kubernetes 服务全网分布图

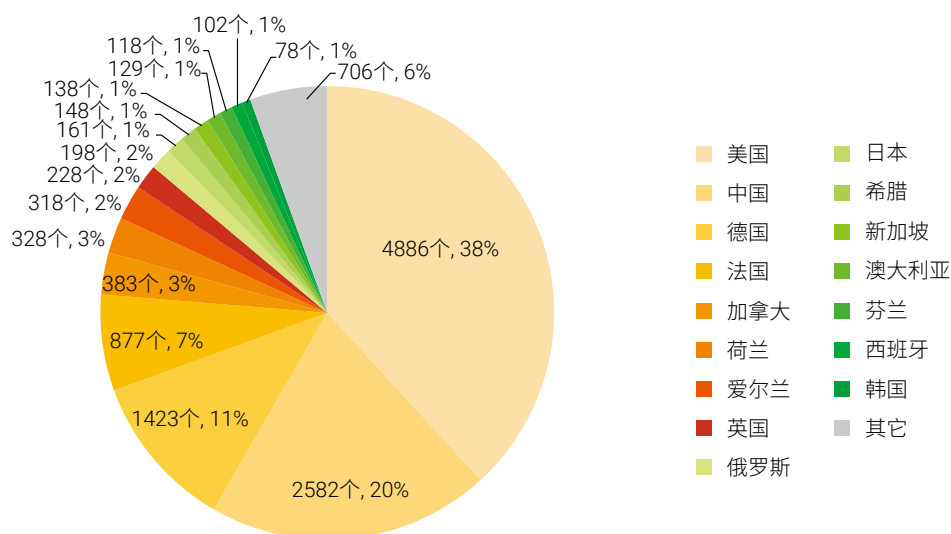
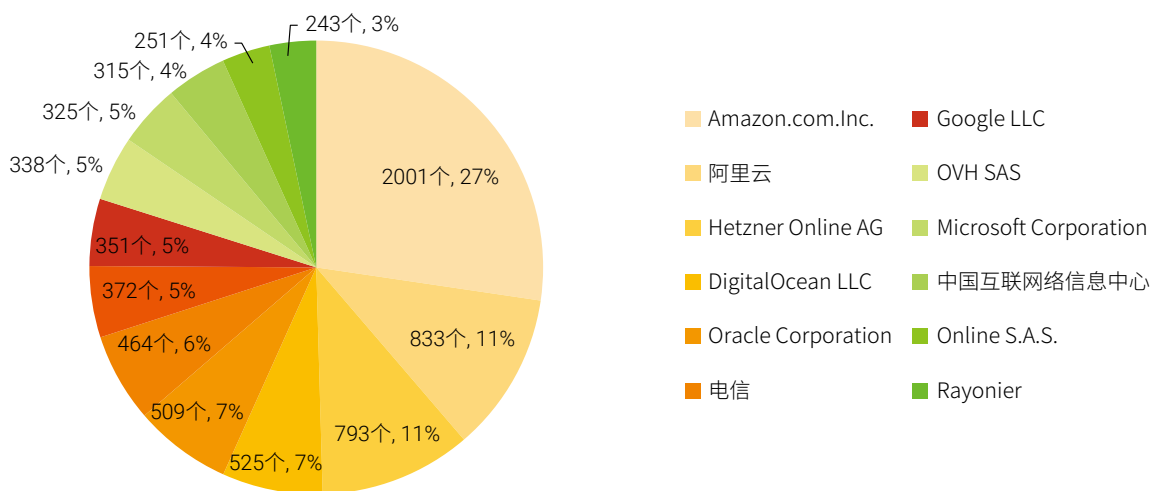


图 3.9 暴露的 Kubernetes 服务的托管服务提供商全球分布图



与 Docker Daemon 远程访问端口 2375 相似，全网范围内暴露 Kubernetes 服务主要分布在美国、中国以及德国，其中美国以 4886 个暴露的服务占比 38% 位居第一，中国以 2582 个暴露的服务占比 20% 位居第二，德国以 1423 个暴露的服务占比 11% 位居第三。这些服务主要部署在亚马逊、阿里云等公有云服务上。

图 3.10 暴露的 Kubernetes 服务国内分布图

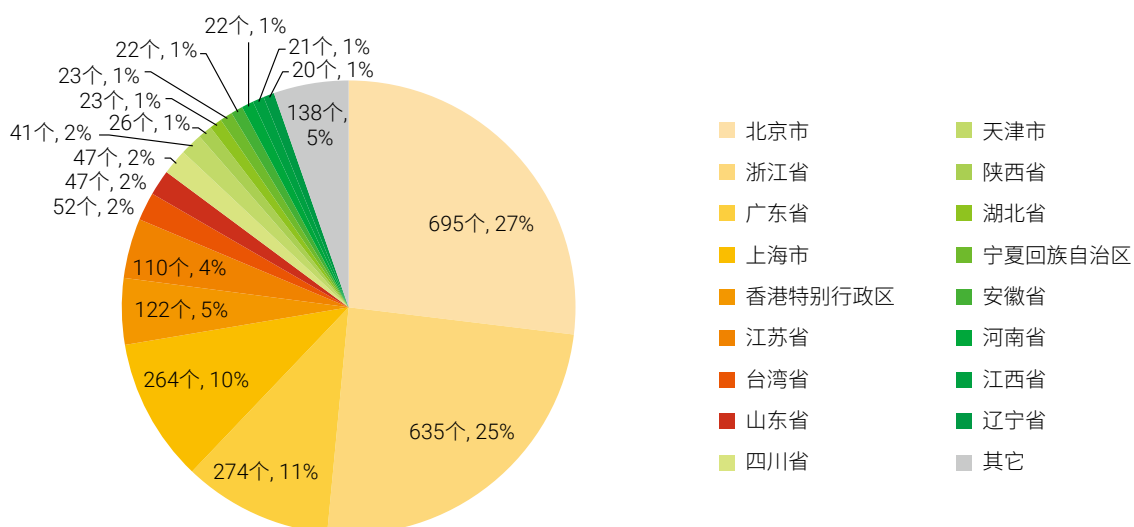
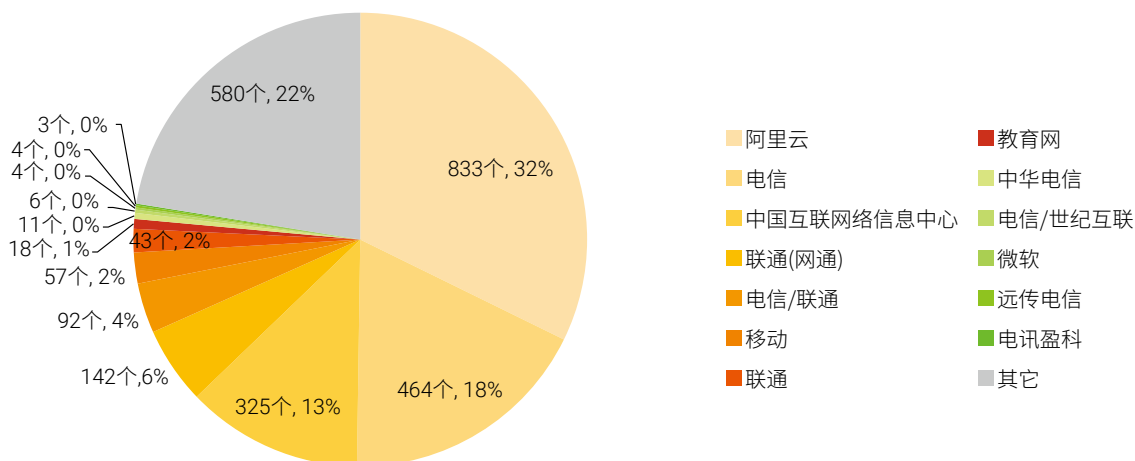


图 3.11 暴露的 Kubernetes 服务的托管服务提供商国内分布



由图 3.10 可见，国内互联网上暴露的 Kubernetes 服务主机主要存在于北京、浙江以及广东等省市。其中北京以 695 个暴露的服务占比 27% 位居第一，浙江以 635 个暴露的服务占比 25% 位居第二，广东以 274 个暴露的服务占比 11% 位居第三。

由上可见，目前国内外的容器编排系统发展迅速，基于其部署特性，互联网上多见于公有云、数据中心，编排系统服务的暴露分布也与容器服务、云服务的发展趋势所匹配。正因为容器和容器编排系统大量部署，相当多的服务被暴露在互联网上，增加了攻击面，其中又有很大一部分服务缺乏基本的安全配置和加固，导致很容易被攻击者利用，破坏系统的可用性和完整性，又有可能被利用对第三方发动各类攻击。

3.1.3 不安全的镜像

开发者通常会在 Docker 官方的 Docker Hub 仓库下载镜像，这些镜像一部分来源于开发镜像内相应软件的官方组织，但还有大量镜像来自第三方组织甚至个人。在整个应用生命周期中，开发人员、测试人员和运维人员会根据不同需求下载并运行镜像，所以在容器运行前进行镜像检查非常重要。

除了 Docker Hub 外，还有大量的第三方镜像仓库，如国内的网易 163^[42]（需注册）、中国科学技术大学 (USTC)^[43]（公共开放）、daocloud^[44]（需注册）、阿里云^[45]等。这些第三方镜像仓库，在提供方便获取镜像的同时，也带来了潜在的安全风险。例如，下载镜像内软件本身是否就包含漏洞，下载的镜像是否被恶意的植入后门，镜像在传输过程中是否被篡改。

（1）镜像使用包含漏洞的软件

有关研究报告^[46]显示，Docker Hub 中超过 30% 的官方镜像包含高危漏洞，接近 70% 的镜像有着高危或中危漏洞。本文从 Docker Hub 中选择评价和下载量较高的 10 个镜像，对其最新版本（latest）采用 Clair^[47] 工具进行了扫描分析，结果如表 3.1 所示。

表 3.1 Docker Hub 中部分镜像漏洞情况统计

镜像	STARS	PULLS	HIGH	MEDIUM	LOW
nginx	8.1K	10M+	3	14	8
ubuntu	7.3K	10M+	0	6	14
mysql	5.8K	10M+	4	7	5
node	5.2K	10M+	68	193	71
redis	4.9K	10M+	6	11	9
postgres	4.7K	10M+	15	32	12
mongo	4.2K	10M+	6	11	9
centos	4.1K	10M+	0	0	0
jenkins	3.4K	10M+	11	25	21
alpine	3.3K	10M+	0	0	0

从表中可以看出，绝大多数的镜像存在高危漏洞，有的镜像高危漏洞数量甚至达到数十个之多。漏洞镜像基本都出现在应用软件，而主流操作系统的脆弱性管理较好，比如 centos 和 alpine 没有扫描出 CVE 漏洞。

（2）攻击者上传恶意镜像

如果有黑客在制作镜像时植入木马、后门等恶意软件，并将恶意镜像上传至 Docker Hub 等公共仓库，那么用户的容器环境从一开始就已经不安全了，后续更没有什么安全可言。

比如 Docker 对镜像不安全的处理管道，Docker 支持三种压缩算法，分别是 gzip、bzip2、xz。gzip 和 bzip2 使用 Go 的标准库，所以相对安全；xz 由于没有使用原生的 Go 去实现，又由于其使用由 C 编写的 XZ Utils 开源项目，所以存在 C 程序恶意写入的可能，一旦被写入会导致执行任意代码漏洞，而只要有一个漏洞，执行 `docker pull` 拉取镜像时就有可能导致整个系统沦陷。

对于特定某文件，一般可以使用杀毒软件进行扫描以确定该文件是否安全，但是目前的杀毒软件并没有能够很好支持镜像的扫描。用户想确认下载的镜像是否安全，只能仔细检查下载的源是否有后门（比如运行镜像，然后在里面安装杀毒软件扫描），并且确认我们的请求指向官方源。

(3) 中间人攻击篡改镜像

如果用户采用非加密的方式从镜像仓库下载镜像，镜像在传输过程中可能被中间人篡改。Docker 在 v1.8 版本中启用内容机制，其中已提供了相应的校验机制来应对这个问题。

3.1.4 容器隔离失效

由于容器与主机共享内核，可能会存在容器隔离失效的安全风险，主要有以下几种场景：

1. 攻击者只要攻破容器操作系统内核，就可访问到主机上的文件系统，或进入其它容器，导致容器隔离失效。
2. 主机的文件系统可以被挂载到多个容器的目录里，那么不同的容器就可以访问同一个目录。例如只要进入某个容器中，就可以通过共同挂载目录，访问其它容器的文件系统，这样可能会引起信息泄露或内容篡改等安全问题。

3.2 安全威胁分析

容器的安全威胁问题主要包括三点，分别是对主机的威胁，对容器的威胁和对承载应用造成的威胁。

3.2.1 容器逃逸攻击

容器逃逸攻击与虚拟机逃逸攻击相似，利用虚拟化软件存在的漏洞，通过容器获取主机权限入侵主机，以达到攻击主机的目的。

具体地，一些 PoC 工具，如 Shocker^[48]，可展示如何从 Docker 容器逃逸并读取到主机某个目录的文件内容。Shocker 攻击的关键是执行了系统调用 `open_by_handle_at` 函数，Linux 手册中特别提到调用 `open_by_handle_at` 函数需要具备 `CAP_DAC_READ_SEARCH` 能力，而 Docker1.0 版本对 Capability 使用黑名单管理策略，并且没有限制 `CAP_DAC_READ_SEARCH` 能力，因而引发了容器逃逸的风险。

3.2.2 容器网络攻击

目前 Docker 总共提供三种不同的网络驱动，分别是桥接网络 (Bridge Network)，MacVLAN，覆盖网络 (Overlay Network)，三种网络驱动都存在着安全风险。

第一种是 Docker 预设的桥接网络驱动，一个 `docker0` 的网桥将所有容器连接该网桥，`docker0` 网桥扮演着路由和 NAT 的角色，容器间通信都会经过容器主机。如果各容器之间没有防火墙保护，攻击者就可以利用主机内部网络进行容器间相互攻击。

第二种是 MacVLAN，这是一种轻量级网络虚拟化技术，MacVLAN 与主机的网络接口连接，MacVLAN 相比于桥接网络驱动加强了与实体网络的隔离性，但是在该网络驱动中，同一个虚拟网络下的容器之间没有进行权限管控，攻击者可以轻易获得容器权限并进行网络攻击。

对此 Docker 官方提供了一些扩展插件来实现对容器权限细粒度的控制，例如网络插件（Network Plugins）可以提供容器间互连网络模型；数据卷插件（Volume Plugins）可以使 Docker 数据卷跨多个主机；验证插件（Authorization Plugins）可以提供基于权限的访问控制等。

第三种是 Overlay Network，主要是利用 VXLAN 技术，在不同主机之间的 Underlay 网络之上再组成新的虚拟网络。这种网络架构在分布式的编排框架中常常用到，在快速构建分布式容器集群的同时，也存在着一些弊端。最为明显的是 VXLAN 网络上的流量没有加密，传输内容很容易被攻击者盗取或篡改。

不过 Docker 使用者可以在设定 IPsec Tunnel 参数时，选择加密来保证容器网络的安全。此外，与前两种驱动一样，Overlay Network 模式下容器间的连接并未有效控制，这也是目前容器网络普遍存在的问题，可能会导致如 ARP 欺骗、嗅探、广播风暴等攻击。

3.2.3 拒绝服务攻击

由于容器在技术实现上基于主机内核，采用共享主机资源的方式，因此面向容器的拒绝服务攻击（DoS）威胁程度更高。例如，默认情况下容器可以使用主机上的所有内存，如果某个容器以独占方式访问或消耗主机的大量资源，则该主机上的其它容器就会因为缺乏资源而无法正常运行。

DoS 攻击可针对任何资源，例如计算资源、存储资源、网络资源等，下面分别以这三种资源进行说明。

（1）计算资源

Fork Bomb¹ 是一个很典型的计算型 DoS 攻击场景，主机内核正常情况下只能支持一定数量的进程，如果某个容器内的进程组新建过多进程，消耗了主机上的所有进程资源，那其它的容器就没有资源来创建新的进程，甚至会危及主机的正常工作。

Fork Bomb 也是自 2015 年到现在 Docker 社区一直讨论的问题，目前最好的方法是限制内存的使用（`--kernel-memory=#M`），但是，当在与加密文件一起使用时可能会偶尔出现问题。

（2）存储资源

在容器技术的实现中，通过 `mount` 命名空间实现了文件系统的隔离。但是文件系统隔离仅仅是一个非常基本的要求。不建议使用 AUFS 做存储驱动，虽然 AUFS 创建出的容器文件系统互相隔离，但是在存储空间方面却没有任何限制。换言之，一个容器如果不断写文件，将会写满存储介质，其它容器将无法执行写操作，导致拒绝服务攻击。

（3）网络资源

DoS 攻击层出不穷，容器内网络带宽耗尽也是其中一种，攻击者使用大量的受控主机向被攻击目标（容器）发送大量的网络数据包，以占满容器的网络带宽，并消耗容器主机的网络数据处理能力，达到拒绝服务的目的。

¹ Fork炸弹(Fork Bomb)在计算机领域中是一种利用系统调用Fork(或其它等效的方式)进行的拒绝服务攻击。与病毒与蠕虫不同的是，Fork炸弹没有传染性，而且Fork炸弹会使对同时执行进程、程序数设限的系统无法执行新程序，对于不设限的系统则使之停止响应。

3.3 容器应用安全威胁

3.3.1 微服务安全

从传统的单体应用到微服务应用，安全一直是人们关注的热点，传统的单体应用由于暴露服务端口少，攻击面较为单一，安全人员也知道常见的攻击点，所以做好相应的防护即可解决大部分问题。

微服务由于将传统的单体应用模块拆分为众多的服务，其端口数量的暴涨必然导致攻击面增多。对于传统的单体应用架构，访问权限、授权机制、隔离审计等只需要做好一道入口的防护即可。

而对于微服务架构来说由于含有多个服务，故每个服务的访问都要进行适当的监控管理和保护。可以想象下，如果在众多的微服务中泄露了某个身份验证令牌或接收了伪造的访问凭证，那整个系统都会陷入威胁中，这些威胁无疑增加了安全防护的难度。

微服务通常由许多服务构成，虽然微服务提倡隔离、轻量、独立开发部署、业务间松耦合等，但有时服务间是需要紧密相联的，比如多个服务间共享数据。这些服务之间的联系在微服务架构中通常是以点对点的形式呈现，随着这些联接的增多，如果其中一个服务因安全漏洞被攻破，那么其它联接的服务也会受到牵连，从而整个系统都会落入攻击者手中。

此外，微服务通常使用容器作为载体，应用被打包成镜像，并以容器的方式分布式地运行在微服务架构中，容器自身有许多安全威胁，比如从容器逃逸宿主机，容器网络攻击等。

3.3.2 DevOps 安全

在网络急速发展的大数据时代，很多企业都贯彻着“敏捷”的思维和行动。例如 DevOps 这种开发和运营的新模式，缩短了软件生命周期各个环节的等待时间，减少了很多重复性、手工性的工作，使得解决问题的成本明显降低，提高了敏捷开发的效率。但敏捷之余忽略安全，这会是一个危险的信号，而且正在不断的被验证着^[49]。

2017 年 11 月 22 日，Uber 发布声明，承认 2016 年曾遭黑客攻击并导致数据大规模泄露。根据这份声明，两名黑客通过第三方云服务对 Uber 实施了攻击，获取了 5700 万名用户数据，包括司机的姓名和驾照号码，用户的姓名、邮箱和手机号。调查发现，Uber 数据泄露的原因竟然是工程师将解锁数据库的安全密钥存储在 GitHub 的一个可以公开访问的页面。

这类由于操作不当引发数据泄漏的事件并不是孤案，尤其值得注意的是，当今云环境和 DevOps 的迅速发展导致安全风险显著地提高了。安全公司 Detectify 的安全顾问 Frans Rosén 曾于 2017 年 7 月 13 日发布一份报告称“网络管理员经常忽略 AWS 访问控制列表（ACL）规则，因服务器错误配置而导致大量数据泄露。”

越来越多的消费者、监管机构和市场发现由此所造成的数据泄漏的代价是高昂的、无法接受的。由于数据泄漏，市场资本中朝夕之间损失数以亿美元计、消费者对企业和机构的信任度下降，在某些情况下，企业负责人的职业生涯可能会受到影响。毫不夸张地说，一些依赖数据生存的企业甚至可能在无意中因为一个密钥存储的疏忽而使整个企业面临灭顶之灾。



4. 安全容器防护

4.1 Linux 内核安全机制	45
4.2 容器服务安全	47
4.3 主机安全	48
4.4 镜像安全	50
4.5 容器网络安全	52
4.6 运行时安全	63
4.7 编排安全	71
4.8 应用安全	74

作为轻量的虚拟化实现方式，容器技术在设计之初就已经做了一些安全考虑，并且构成了容器安全重要的防护基础。本章将就容器面临的安全风险与威胁，介绍常见的防护思路和防护方法。

4.1 Linux 内核安全机制

基础安全主要指 Linux 内核相关功能模块实现，包括容器资源的隔离、管控等。容器技术，无论是 Docker 还是 LXC，在初始设计时，都具有类似的安全考虑。例如通过内核命名空间（Namespace）构建一个相对隔离的运行环境，保证了容器之间互不影响；通过控制组（CGroups）对 CPU、内存、磁盘 I/O 等共享资源进行隔离、限制、审计等，避免多个容器对系统资源的竞争。这些 Linux 内核模块功能，构成了容器的基本安全保障。

4.1.1 内核命名空间

内核命名空间是 Linux 内核一个强大的特性。新创建一个容器时，后台会为其创建一组命名空间和控制组的集合。命名空间为容器提供了最原始也是最简单的隔离方式。保证一个容器中运行的进程看不到或者影响不到运行在另一个容器中的进程或者是容器主机的进程。命名空间提供了 PID 命名空间、网络命名空间、IPC 命名空间、mnt 命名空间、uts 命名空间和用户命名空间等。

例如，PID 命名空间实现了不同用户之间进程的隔离，保证不同的命名空间中可以有相同的 pid，同时可以方便地实现容器的嵌套。容器中进程的交互，采用了 Linux 常见的进程交互方法 IPC（Inter-Process Communication），包括信号量、消息队列和共享内存等，需要在 IPC 资源申请时，加入命名空间信息，保证每个 IPC 资源有一个唯一的 32 位 Id。

容器网络的隔离通过网络命名空间实现，每个网络命名空间拥有独立的网络设备、IP 地址、路由表等。每个容器也都拥有自己的网络协议栈，不同的容器不会获取对其它容器的套接字或接口的访问权限。

Linux 内核在 2.6.26 之后引入了内核命名空间，这也就意味着自 2008 年 7 月 2.6.26 版本内核发布之后，命名空间已经已经在大量的生产系统得到使用和验证。其设计和实现可以认为较为成熟。

4.1.2 控制组

控制组 CGroups 是 Linux 内核的另一个重要特性，主要用来实现对资源的限制和审计等，该技术最早由 Google 在 2006 年提出，Linux 内核自 2.6.24 版本开始进行支持。

在容器技术的实现中，CGroups 是一个重要的组成部分，它提供了多种度量标准，来确保每个容器获得公平的 CPU、内存和 I/O 等资源，对容器资源进行限制和审计。同时限制某个容器的最大资源使用量，防止其耗尽资源致使系统性能降低。

CGroups 虽然不能实现容器间的隔离，但是其对资源的审计和限制，对于缓解拒绝服务攻击有着重要的作用，尤其是像公有云和私有云的 PaaS 服务在面对多租户时，保证每个租户容器的正常运行，防止恶意租户滥用资源。

4.1.3 内核功能限制（Capabilities）机制

通常 Linux 系统上众多的服务进程以 root 用户权限运行，比如 SSH、Cron、Syslog、硬件管理工具、网络配置工具等，一旦服务被攻击者攻破，攻击者就有系统最高权限进行破坏。Linux 内核功能提供了更细粒度的权限控制，例如挂载、访问文件系统，以及内核模块加载等。

容器服务可利用 Linux 内核功能的机制，在多数情况下避免使用真正的主机 root 用户权限。因此容器可以运行在一个内核功能集合的约束下，比如在容器创建的时候，可以拒绝所有的挂载（mount）文件系统、访问原始套接字、创建新设备节点、以及更改文件所有者 / 属性等文件系统操作。这样，即使容器遭到攻击者入侵，攻击者也是很难在容器内部对宿主机做恶意操作。

默认情况下，Docker 采用白名单方式实现功能限制的，也就是说，白名单之外的功能会全部被禁用，在 Linux 手册中可以查看完整的可用功能列表^[50]。

不过运行 Docker 容器时有一个风险：容器采用默认内核功能和挂载可能会导致容器之间的隔离问题，此外，攻击者通过主机系统漏洞对容器产生威胁。因此 Docker 允许通过使用非默认配置文件添加和删除内核功能，来增加其安全性。对于用户，应该明确其所需的具体功能，删除不必要的内核功能。

4.1.4 其它内核安全机制

内核功能是现代 Linux 内核提供的众多安全特性之一，除此之外，还可以利用其它常用的安全机制来加固和防护 Docker 容器，比如 TOMOYO^[51]，AppArmor^[52]，SELinux^[53]，Grsec^[54] 等。

比如使用 Grsec 和 PAX 运行内核可在编译和运行时，基于地址随机化等技术增加安全检查，这样就避免了许多漏洞利用。由于这些安全特性属于宿主机系统范畴，独立于容器，因此不需要对容器进行特殊的配置。

另外，如果使用的发行版操作系统带有 Docker 容器的安全模板，则可以直接使用。例如，Docker 官方团队发布了一个可与 AppArmor 配合使用的模板，Red Hat 也提供了适用于 Docker 的 SELinux 策略。这些模板提供了一个额外的安全网，用户可以使用特定的访问控制机制来定义自己的策略。

4.1.4.1 SELinux 安全加固

首先介绍强制访问控制（Mandatory Access Control，MAC），它是指计算机系统根据使用系统机构事先确定的安全策略，对用户的访问权限进行强制性的控制，也就是说系统独立于用户行为强制执行访问控制。

SELinux 最早由美国国家安全局（NSA）发起，是一种强制访问控制的实现，它是 Linux 历史上最杰出的安全子系统。在这种访问控制体系的限制下，进程只能访问那些在它的任务中所需的文件。对于目前可用的 Linux 安全模块来说，SELinux 功能最全面，而且测试最充分，它是基于对 MAC 20 年的研究基础之上建立的。

使用方法：启动 Docker 服务时需添加 SELinux 选项。

```
# docker daemon --selinux-enabled = true
```

当运行 shocker¹ 时：

```
# docker run --rm -ti --cap-add=all shocker bash
```

shocker 的代码在暴力破解的过程中循环遍历找出 `/etc/shadow`，但是读取时被 SELinux 阻止了，返回信息为：

```
#open: Permission denied
```

可见 SELinux 安全加固确实有效。

1 Github 上的项目 Shocker，它描述了怎样逃逸 Docker 容器并读取到 Host 的 `/etc/host` 文件的内容，完整代码为：<http://stealth.openwall.net/xSports/shocker.c>

4.1.4.2 AppArmor 安全加固

AppArmor 也是一种 MAC 控制机制，其主要作用是设置某个可执行程序的访问控制权限，可以限制程序读 / 写某个目录 / 文件，打开 / 读 / 写网络端口等。

Docker 服务在启动过程中会判断当前内核是否支持 AppArmor，若支持，就创建默认的 AppArmor 配置文件 `/etc/apparmor.d/docker`，并应用这个配置文件。启动容器时，在初始化过程中 Docker 会使用相应的 AppArmor 配置作用于容器。也可以使用 `-security-opt` 选项来指定作用于容器的 AppArmor 配置文件。

制定一个 AppArmor 规则，并应用到容器上。首先拷贝一个模板。

```
# cp /etc/apparmor.d/docker /etc/apparmor.d/container
```

使用方法：修改 `/etc/apparmor.d/container` 配置文件，加入一行 `deny /etc/hosts rwkx`，用该规则运行 shocker。

```
# apparmor_parser -r /etc/apparmor.d/container
# docker run --rm -ti --cap-add=all --security-opt apparmor:container-default shocker bash
```

返回的结果同样为 `open: Permission denied`，攻击依然不奏效，“`deny /etc/hosts rwkx`” 该条规则限制了容器内的其它程序对 `/etc/hosts` 的获取。

容器技术跟虚拟化技术相比，没有了 Hypervisor 这一层，即容器的资源管理没有 VMM（Virtual Machine Manager），而完全依赖于主机的操作系统，那么如何保证容器是按照其规范正确的使用，成为了容器安全的一个重要的前提。

Docker 公司与美国互联网安全中心（Center for Internet Security, CIS）合作，发布了一份详尽的基准配置文档^[55]，该基准文档从容器主机、Docker 守护进程、镜像配置、运行配置等方面，给出了参考的配置建议。

4.2 容器服务安全

容器的管理和编排服务的安全直接影响了容器控制平面的安全，以 Docker 为例，Docker 守护进程（Docker Daemon）的配置合理与否，将在一定程度上影响 Docker 的安全性。建议在守护进程启动时进行以下配置：

（1）限制容器间网络流量

默认情况下，同一主机上的容器之间是允许所有通信的，即黑名单机制，可根据业务需求添加访问控制规则。为了进一步限制容器间通信，可以采用白名单机制（给 Docker 守护进程传递命令行参数 `--icc = false`），即默认容器间的通信是禁止的，只有显示添加规则才可以访问。此外，也可将容器加入自定义的网络，也可以限制与其它网络的容器间通信。

（2）设置集中和远程日志管理

容器的生命周期很短，且业务变化很快，为了完成该场景下的安全分析和取证，需要将日志存放在远程集中的平台中，可以通过 `docker --log-driver=syslog --log-opt syslog-address=tcp://192.xxx.xxx.xxx` 进行设置。

(3) 允许 Docker 更改 Iptables 规则

Docker 守护进程会根据用户的网络设置，自动对 Iptables 进行必要的更改，建议允许 Docker 守护进程自动更新 Iptables 配置，可以通过 `docker --iptables = true` 启动 Docker。

(4) 不使用 AUFS 作为存储驱动程序

AUFS 是 Linux 较旧的存储驱动程序，可能会引发一些严重的内核崩溃等问题，在许多最新的 Linux 内核发行版中，不再对 AUFS 进行支持。可以通过 `docker --storage-driver devicemapper` 将存储驱动设置为 Device Mapper。默认情况下，Docker 在大多数的平台采用 Device Mapper 作为存储驱动，这可能会因不同的操作系统而不同，应首选操作系统支持最佳的存储驱动。

(5) 使用默认 CGroups

Docker 通过 CGroups 对容器的资源进行限制，可以通过 `docker --cgroup-parent = /foobar` 进行设置。

对于守护进程的配置，除了上述命令行的方式，一般操作是修改“docker.service”文件，文件路径通常为 `/lib/systemd/system/docker.service`，不同 Linux 发行版本可能路径不同，修改完成后重启 Docker 守护进程就有以上命令属性了。

(6) 应遵循在 Docker 守护进程级别设置 ulimit

ulimit 是一种 Linux 系统的内建功能，它具有一套参数集，用于为它生成的 Shell 进程及其子进程的资源使用设置限制。其支持限制的資源包括：所创建的内核文件的大小、进程数据块的大小、Shell 进程创建文件的大小、内存锁住的大小、常驻内存集的大小、打开文件描述符的数量、分配堆栈的最大值、CPU 时间、单个用户的最大线程数、Shell 进程所能使用的最大虚拟内存。同时，它也支持硬资源和软资源的限制。

设置系统资源控制可以防止资源耗尽带来的问题，比如 Fork 炸弹，有时候合法的用户和进程也可能造成过度的使用系统资源，导致系统资源耗尽。因此在 Docker 服务需要设置默认的 ulimit 值，实现相应的资源控制。

(7) 使用可信仓库

默认情况下，Docker 进程只允许 `push` 和 `pull` 配置了可信证书的镜像仓库的镜像，除非是自己内部的不可信的仓库否则不建议开启此配置项。

4.3 主机安全

4.3.1 主机基本安全加固建议

容器与宿主机共享操作系统内核，因此宿主机的配置对容器运行的安全有着重要的影响，比如宿主机中安装有漏洞的软件可能会导致任意代码执行风险，端口无限制开放可能会导致任意用户访问风险，防火墙未正确配置会降低主机的安全性，sudo 的访问权限没有按照密钥的认证方式登录可能会导致暴力破解宿主机。

从安全性考虑，容器主机应遵循如最小安装化，不应当安装额外的服务和软件以免增大安全风险；配置交互用户登陆超时时间；关闭不必要的数据包转发功能；禁止 ICMP 重定向；配置可远程访问地址范围；删除或锁定与设备运行、维护等工作无关的账号、重要文件和目录的权限设置；关闭不必要的进程和服务等安全加固原则。

4.3.2 容器相关安全加固建议

(1) 为容器的存储分配单独的分区

默认情况下，所有 Docker 相关文件存储在 `/var/lib/docker` 目录下，需要尽可能地为容器创建独立的分区以保证其安全性。对于新安装的 Docker，可以通过以下命令实现审计：`grep /var/lib/docker /etc/fstab`。

(2) 宿主机安全加固

保证宿主机符合相应的安全规范，对其进行有效的漏洞管理和配置管理。

(3) 更新 Docker 到最新版本

由于 Docker 软件会不断地发布更新，旧的版本可能存在安全漏洞。因此需要保证已发现的 Docker 软件漏洞尽可能得到修复并且定期对 Docker 版本进行风险评估。

(4) 守护进程的控制权限

Docker 的守护进程需要 root 权限，对于添加到“docker”用户组的用户提供了完整的 root 访问权限。因此在容器主机上，要严格限制“docker”用户组内的用户，删除一切不受信任的用户。

(5) 对 Docker 守护进程进行审计

在容器上，除了需要对常规的 Linux 文件系统和系统调用进行审计之外，还需要对 Docker 守护进程的活动和使用情况进行审计。默认情况下，是不会对其进行审计的，可以通过 `auditctl -w /usr/bin/docker -k docker` 添加审计规则，也可以通过修改 `/etc/audit/audit.rules` 文件进行规则更新。

在进行 Docker 守护进程审计时，会产生大量的日志文件，要确保定期的进行日志归档，建议创建一个单独的审计分区进行日志存储，以避免日志文件写满根系统影响正常业务。

(6) 对 Docker 相关的文件和目录进行审计

除了对 Docker 守护进程进行审计，还需要对 Docker 相关的文件和目录进行审计，例如 `/var/lib/docker`（包含有关容器的所有信息）、`/etc/docker`（包含 Docker 守护进程和 Docker 客户端之间 TLS 通信的所有密钥和证书）、`docker.service`（Docker 守护进程运行参数配置文件）、`docker.socket`（守护进程运行 socket）、`/etc/default/docker`（支持 Docker 守护进程各种参数）、`/etc/default/daemon.json`（支持 Docker 守护进程各种参数）、`/usr/bin/docker-containerd`、`/usr/bin/docker-runc`（Docker 依赖于 containerd 和 runC 来生成容器）。

具体审计方式与守护进程的审计一致，可以通过配置文件或命令行的方式添加审计规则。

4.4 镜像安全

作为容器运行的基础，容器镜像的安全在整个容器安全生态中占据着重要位置。容器镜像是由若干层镜像叠加而成的，通过镜像仓库分发和更新的。因此，可以从镜像构建安全、仓库安全以及镜像分发安全三方面介绍镜像安全。

4.4.1 镜像构建安全

镜像构建的方式通常有两种：基于容器直接构建或基于 Dockerfile 构建。但建议所有的镜像文件应该由 Dockerfile 来创建，因为基于 Dockerfile 构建的镜像是完全透明的，所有的操作指令都是可控和可追溯的。

镜像构建存在的风险项通常包括：

1. `pull` 的基础镜像并不是由可信的组织 and 人员发布，镜像本身存在后门或者其它风险项。
2. 在 Dockerfile 中存储敏感信息。例如配置服务时使用明文固定密码或凭证等。
3. 安装不必要的软件扩大了攻击面。

针对以上问题可以从下面几方面来加固镜像构建安全：

（1）验证镜像来源

为了保证镜像内容的可信，建议开启 Docker 的内容信任机制。内容信任机制为向远程镜像仓库发送和接收的数据提供了数字签名功能，这些签名允许客户端验证镜像标签的完整性和发布者。默认情况内容信任机制是被禁用的，可以通过执行下面指令完成或者在 Docker 的配置文件中配置。

```
# export DOCKER_CONTENT_TRUST=1
```

（2）镜像轻量化

只安装必要的软件包，不仅在提高容器性能方面有很大帮助，更重要的是减少了攻击面。

（3）正确使用镜像指令

在构建镜像时要选择恰当的指令，若需引入外部文件，在 Dockerfile 中能用 `COPY` 指令就不要使用 `ADD` 指令，因为 `COPY` 指令只是将文件从本地主机复制到容器文件系统，`ADD` 指令却可以从远程 URL 下载文件并执行诸如解压缩等操作，这可能会带来从 URL 添加恶意文件的风险。

（4）敏感信息处理

尽管 Docker 为用户分配了只读权限，但有时用户仍需要小心容器中存储的数据，例如 Dockerfile 中不能存储密码，令牌，密钥和用户机密信息等，即使在创建好容器后再删除这些数据也会造成风险，因为镜像的历史记录中仍能检索到这些数据。

推荐使用 Kubernetes 和 Docker Swarm 的加密管理功能，可以有效的对信息进行加密，以加密格式存储，并且在查找时只能由授权的用户去解密。

4.4.2 镜像仓库安全

4.4.2.1 公共仓库安全

Docker Hub 是目前最大的容器镜像仓库，前述章节提到，Docker Hub 中超过 30% 的镜像包含高危漏洞。因此，在享受 Docker Hub 带来的便利时，也应确保下载镜像的安全性。

1. 在选择镜像时，应使用官方发布最新版本的镜像，并保持定时更新。
2. 下载的镜像要经过漏洞扫描评估。
3. 对于提供服务的镜像，不仅要从操作系统层面进行扫描，还要从应用层面进行扫描。
4. 对于提供了公开 Dockerfile 的镜像优先选择自己构建，可避免镜像后门的植入，保证镜像构建过程的可控。

4.4.2.2 私有仓库安全

(1) Docker Registry

Docker Registry^[56] 是 Docker 官方提供构建私有镜像仓库的开源工具，开发者可快速构建自己的私有仓库。Docker Registry 的安全性可以从两方面考虑：

一方面是 Docker Registry 自身的安全性，例如在使用时要配置相应的安全证书。

另一方面是 Docker 客户端与 Docker Registry 交互过程的安全性，即实现用户访问权限控制。对于暴露在互联网上的私有仓库，一般只对特定组织开放存取镜像权限，那么仅验证 Registry 自身的证书是不够的，还需要配置密码或双向 SSL 机制来验证与仓库进行交互的 Docker 客户端的身份有效性。

(2) VMware Harbor

将 Harbor 部署在生产环境需要注意：需要启用 HTTPS 协议，不能使用 `harbor.cfg` 中默认密码；Harbor 在用户身份登录时没有做防暴力破解机制，存在密码被破解的风险，生产环境中需要通过修改源码添加防暴力破解机制；严格控制挂载卷权限，默认情况下是读写（rw）模式，可选的模式有读写（rw）和只读（ro）。

4.4.3 镜像扫描

容器都是由本地存储的镜像快速启动运行的，那么镜像的安全性直接关系到容器安全。除了前述提及容器仓库中镜像安全性的问题外，本地构建的镜像也会引入第三方库，造成安全风险。所以，对下载的镜像和本地构建的镜像进行安全扫描，就显得尤其重要。

目前比较流行的镜像扫描引擎有 Docker Security Scanning（不开源）、Clair（开源）和 Anchore（开源）等。

镜像检测的核心目前仍然是已知系统 CVE 检测。扫描器获取到镜像后，将它分离成相应的层和软件包（Package）。然后这些 Package 将与多个 CVE 数据库 Package 的名称和版本进行对比，从而判定是否存在漏洞。

还有一些通过扫描镜像中的环境变量、操作命令以及端口开放信息来识别恶意镜像的方案，但仍然需要使用者自己基于结果来判断，还不能直接给出一个明确的结果。

总体来讲，现有成熟的检测方案还是有一些局限性，仍有很大的发展改进空间。

4.4.4 镜像传输安全

容器镜像在下载和上传时需保证完整性和秘密性，以下建议有助于抵御如中间人攻击等威胁：

(1) 数字签名

上传者主动给要上传的镜像签名，下载者获取镜像时先验证签名再使用，防止其被恶意篡改。

(2) 用户访问控制

敏感系统和部署工具（注册中心、编排工具等）应该具备有效地限制和监控用户访问权限的机制。

(3) 尽可能使用支持 HTTPS 的镜像仓库

为避免引入可疑镜像，用户谨慎使用 `--insecure-registry` 选项连接来源不可靠的 HTTP 镜像仓库。

4.5 容器网络安全

4.5.1 网络自身安全机制

计算机网络基本的两大防护手段是隔离和访问控制，本节分别以 2.3 节中的 Bridge 模式的主机网络和 Overlay 集群网络为例，讨论上述安全机制的实现。总的来说，在开源的方案中，容器的网络隔离主要借助网络命名空间和 Iptables 来实现；而访问控制则是通过 Iptables 来实现。

4.5.1.1 Bridge 模式的网络隔离和访问控制

在 Bridge 模式的主机网络中，从 Docker 的设计原则上看，只要当两个容器连接到同一个网桥，这两个容器就可以互相访问¹，并没有相应的访问控制或隔离机制。如果要实现两个容器隔离，则需要新建桥接网络，并将这两个容器放在不同的桥接网络中。

具体地，首先创建一个桥接网络 *test*：

```
# docker network create --subnet 102.102.0.0/24 test
c11c01a07ed0ca3f4cdddee55e3e058e79c334d516b3a49dd3e56b86a4ff9302

# ifconfig | grep 102.102.0. -B 1
br-bff064219957 Link encap:Ethernet HWaddr 02:42:69:46:0b:21
inet addr:102.102.0.1 Bcast:102.102.0.255 Mask:255.255.255.0
```

¹ 默认情况下 Docker 守护进程启动参数 `--icc=true`，表示容器间默认的通信策略为放行

可以获得桥接网络 *test* 对应的网桥名称为 *br-bff064219957*。当新网络建立后，Docker 会在 Iptables 中的 *DOCKER-ISOLATION* 链新增一些 *DROP* 规则，来阻断该网络与其它网络的双向流量，从而实现网络之间的隔离。

```
# iptables -t filter -L -v
Chain DOCKER-ISOLATION (1 references)
pkts bytes target prot opt in out source destination
0 0 DROP all -- docker_gwbridge br-bff064219957 anywhere anywhere
0 0 DROP all -- br-bff064219957 docker_gwbridge anywhere anywhere
0 0 DROP all -- docker0 br-bff064219957 anywhere anywhere
.....
```

在零信任（Zero Trust）模型下，不同容器间默认应该是不能通信的，此时可以设置 Docker 守护进程启动参数 *--icc=false*，启动后 Iptables 的 *FORWARD* 链默认策略为丢弃，然后运行时根据业务需要再添加访问控制策略。在云工作载荷防护（Cloud Workload Protection）中，这样基于白名单策略在部署关键业务的环境中是非常重要的原则，可最小化服务的暴露面并限制未知业务的连接，从而减少被攻破的可能。

```
# iptables -nL
Chain FORWARD (policy DROP)
DROP all -- 0.0.0.0/0 0.0.0.0/0
.....
```

4.5.1.2 集群模式的网络隔离和访问控制

本节以 Docker Swarm 为例，介绍集群模式下的容器网络安全机制。简单而言，Docker Swarm 的 Overlay 网络是采用 IETF 标准的 VXLAN 方式，从而实现不同子网的隔离。我们通过下面的例子来说明 Docker Swarm 的 Overlay 网络的隔离和访问控制机制。

（1）建立集群环境

首先，准备一个两节点的集群：

```
# docker node ls
ID HOSTNAME STATUS AVAILABILITY MANAGER STATUS ENGINE VERSION
otsycjte53ve20byk79aukw14 * node1 Ready Active Leader 18.03.1-ce
icd266rw1mpa47mghng1jvd7b node2 Ready Active 17.12.1-ce
```

(2) 创建 Overlay 网络

分别创建名为 *test0* 和 *test1* 的两个 Overlay 网络：

```
# docker network create -d overlay test0
vn86l7s11gfj9ttnzexhgl0yh
# docker network create -d overlay test1
w9p3tfmr09c6gw1to0guglfc0
```

```
# ls -l /run/docker/netns
total 0
-r--r--r-- 1 root root 0 May 28 00:56 1-3cv9ry2rsk
-r--r--r-- 1 root root 0 Jun 1 14:15 1-vn86l7s11g
-r--r--r-- 1 root root 0 May 9 19:08 default
-r--r--r-- 1 root root 0 Jun 1 14:17 1-w9p3tfmr09
-r--r--r-- 1 root root 0 May 28 00:56 ingress_sbox
.....
```

可见，Docker 为每个 Overlay 网络新建了一个 Namespace，Overlay 类型网络的 namespace 的名字前都会添加 “1-”，后面的一串数字对应于 Overlay 网络的 Id 字符串的前缀。如 *test0* 网络对应的 Namespace 是 *1-vn86l7s11g*，*test1* 网络对应的 Namespace 是 *1-w9p3tfmr09*，Ingress 网络对应的为 *1-3cv9ry2rsk*，这些命名空间的名称在不同主机上是相同的。

(3) 创建服务

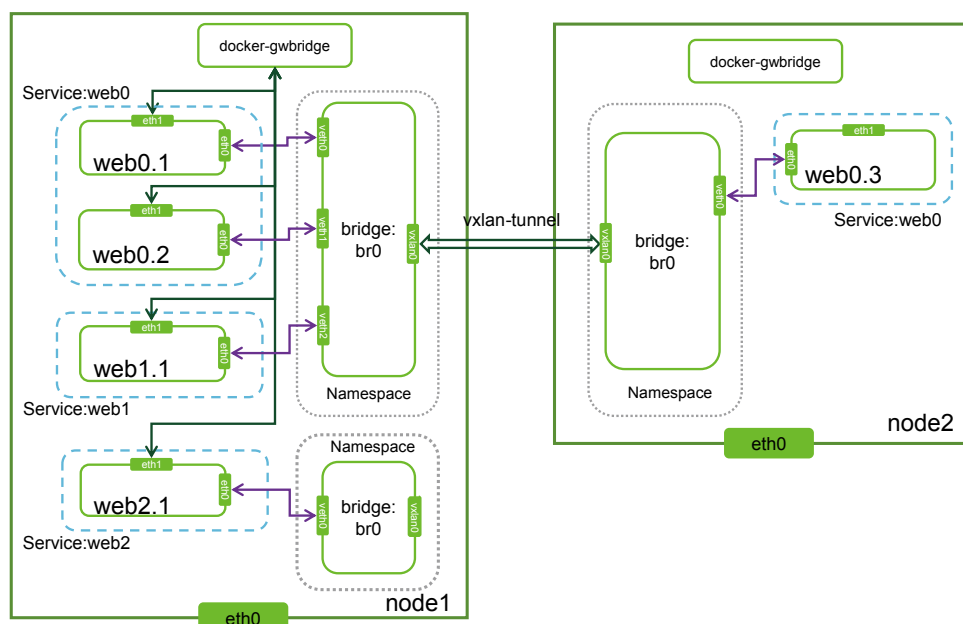
分别创建 web0、web1 和 web2 服务，其中 web0 和 web1 共享 *test0* 网络，web2 独享 *test1* 网络¹。

```
# docker service create --name web0 --network test0 nginx:alpine
ugeorthtdyrf6axaxdigseyru
# docker service create --name web1 --network test0 nginx:alpine
i4rg9tdkuhv2maf9wzwdkclog
# docker service create --name web2 --network test1 nginx:alpine
wk34tw25bvdu9zxdkbsfxv1w
```

¹ 所建服务对应的容器除了 lo 接口外，还有两个网络接口，其中 eth0 连接到所创 Overlay 网络 test0 或 test1 的网桥接口，用于完成跨主机同网络不同容器间的通信，eth1 是连接到 docker-gwbridge 桥接网络，主要用于完成容器和主机通信。如果在创建 Overlay 网络时使用了 `-internal`，则通过 `docker service create --network` 创建的服务所对应的容器，除了 lo 接口外，只有一个网络接口，其中 eth0 连接到所创 Overlay 网络的网桥接口。

设其中 web0 的副本数为 3，web1 和 web2 的副本数分别为 1。则整个集群的网络拓扑图如下所示：

图 4.1 集群网络拓扑



其中 web0.1、web0.2、web0.3 为服务 web0 创建的三个副本容器，web1.1 和 web2.1 分别为服务 web1 和 web2 创建的副本容器。其中 web0.1 的完整名称为 web0.1.u800qsqymguo9avo72w2o3ci0，为了简单起见后文省略了容器实例的完整后缀。

当建立好环境后，可通过服务找到与其连接的网络，以 web0 为例：

```
# docker service inspect web0
...
"VirtualIPs": [
  {
    "NetworkID": "vn8617s11gffj9ttznzexhgl0yh"
  },
  {
    "Addr": "10.0.0.10/24"
  }
]
...
```

可见该服务连接了 *test0* 网络，所以该服务里的副本容器也至少会连接这个网络，如果服务对外提供服务，还会连接 Ingress 网络。如查询 *test0* 网络，可找到其连接的容器：

```
# docker network inspect test0
[
  {
    "Name": "test0",
    "Id": "vn8617s11gfj9ttznzexhgl0yh",
    "IPAM": {
      "Driver": "default",
      "Options": null,
      "Config": [
        {
          "Subnet": "10.0.0.0/24",
          "Gateway": "10.0.0.1"
        }
      ]
    },
    "Containers": {
      "759bbdf216d801615ff90578dd5a828e53449990a9016b82fdee1689cb7d448f": {
        "Name": "web0.1.u800qsqymguo9avo72w2o3ci0",
        "EndpointID": "b03c9a96d81d91301594951fcc08ec0765102b37810c77ec8cb4943bda4e9868",
        "MacAddress": "02:42:0a:00:00:06",
        "IPv4Address": "10.0.0.6/24",
      },
      "791de588518714ae1b3140956918c5d5adcae151f10ef3c9b5808ff0638ab357": {
        "Name": "web0.1.n9pjx00w99fwqch01gaiew90",
        "EndpointID": "8bfbb44db8e81734676d4756d7152a5ea3f57d7aae987600e238b5a70c15d670",
        "MacAddress": "02:42:0a:00:00:04",
        "IPv4Address": "10.0.0.4/24",
      },
      "e6c02f73a9c72d23fcd5c17aa265adf1de56d1c933f396e6995b066c0b333f11": {
        "Name": "web0.3.rmmfjc7jmr8grld6pwkm5e0q",
        "EndpointID": "d223943dc20da5c57bcd19d9f2b743cd59cad00b6a84eae44048937ef3780324",
        "MacAddress": "02:42:0a:00:00:05",
        "IPv4Address": "10.0.0.5/24",
      },
      "e9fc84b0d736107cec501436d08d4911dd4a516bfcce3e5b95016913ff2b8521": {
        "Name": "web1.1.6f9chzfd711nmjenc6bgg109x",
        "EndpointID": "703622e4d0cd0ceca52c6b10a2c5451e6767845147625a21d18a7cf99b7cef56",
        "MacAddress": "02:42:0a:00:00:07",
        "IPv4Address": "10.0.0.12/24",
      },
    },
    "Peers": [
      {
        "Name": "17a9fb8e9cd2",
      }
    ]
  }
]
```

可见，Docker Swarm 在主机之间建立 VXLAN 隧道，从 *test0* 网络的信息可见，其在 192.168.19.11 和 192.168.19.12 两个主机组成的 Underlay 网络之上，构造了一个 10.0.0.0/24 的 Overlay 子网，网关为 10.0.0.1（即命名空间 *1-vn86l7s11g* 的 *br0* 的网络地址）。

此外，当服务 *web0* 和 *web1* 连接到同一个网络 *test0* 时，其容器副本均连接到了同一个子网，从网络对应的命名空间的网络接口 VLAN 标识可见，网桥上的各个网络接口没有设置 VLAN，在同一主机内部的同一网络的容器均可互相访问，没有隔离措施。

```
# s1-vn86l7s11g bridge vlan
port      vlan ids
br0       None
vxlan0    None
veth0     None
veth1     None
# nsenter --net=/var/run/docker/netns/1-vn86l7s11gbridge fdb show dev vxlan0
22:71:ed:59:a9:59 master br0 permanent
02:42:0a:00:01:09 dst 192.168.19.12 link-netnsid 0 self permanent
02:42:0a:00:01:08 dst 192.168.19.12 link-netnsid 0 self permanent
02:42:0a:00:01:06 dst 192.168.19.12 link-netnsid 0 self permanent
```

在 OpenStack 的计算节点中，在 VXLAN 隧道相连的网桥上，会通过 VLAN 对不同租户的子网进行隔离，而在 Docker Swarm 中，由上可知并非通过 VLAN，而是借助网络命名空间，将不同的网络进行隔离。

如服务 *web0* 的容器副本连接到了网络 *test0*，即数据包从容器进入了命名空间 *1-vn86l7s11g* 中的网桥 *br0*，只能发向连接网络 *test0* 的其它容器副本（即本主机或其它主机的命名空间 *1-vn86l7s11g* 中 *br0* 所连接的容器），而无法发向网络 *test1* 连接的容器。可以通过下面验证：

```
# docker exec web0.1 ping 10.0.0.12 -c 1
PING 10.0.0.12 (10.0.0.12): 56 data bytes
64 bytes from 10.0.0.12: seq=0 ttl=64 time=0.250 ms
--- 10.0.0.12 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.194/0.194/0.194ms

docker exec web0.1 ping 10.0.1.5 -c 4
PING 10.0.1.5 (10.0.1.5): 56 data bytes
--- 10.0.1.5 ping statistics ---
4 packets transmitted, 0 packets received, 100% packet loss
```

（4）集群下容器的访问控制

如前所述，Docker Swarm 中，相同 Overlay 网络的容器之间默认可以互相访问，不同 Overlay 网络之间的容器不能互相访问。容器之间是没有可定制的访问控制策略，如果需要显式地应用访问策略需要手动添加。

我们新建一个对外暴露的服务：

```
# docker service create --name web5 --network test1 --replicas 3 -p 3000:80 nginx:alpine
c3ql2e1e0g7xbxksfz3pjlrgb
```

因为创建服务时选择了 *-p* 参数，则此服务所对应的容器会创建 4 个网络接口，其中：

eth0 连接到 Ingress 网络，Ingress 是用来完成外部网络到容器的连接；

eth1 连接到 docker-gwbridge 网络，docker_gwbridge 提供了容器到外部网络的通信，例如在容器里执行 `ping www.baidu.com` 则流量会经过 default-gwbridge 网络；

eth2 连接到 test1 网络，主要完成跨主机服务间不同容器副本间的通信。

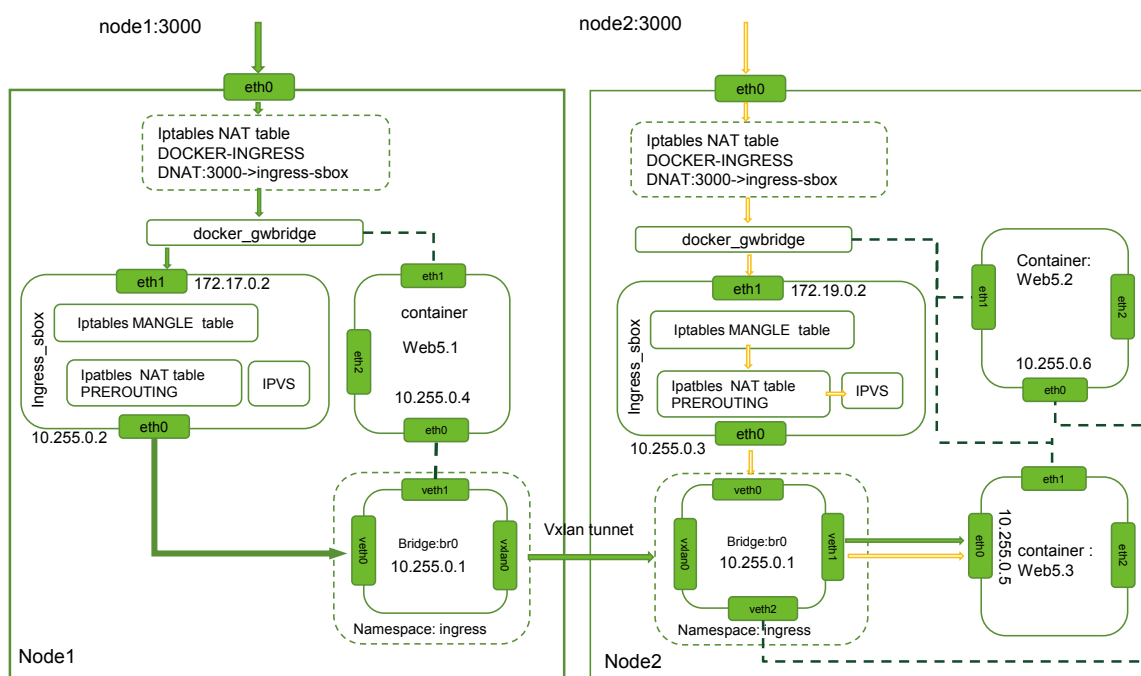
服务对外暴露 3000 端口，而容器监听 80 端口，外部通过 3000 端口访问服务是由端口映射来实现：

```
# iptables -t nat -n -L | grep -A 4 'DOCKER-INGRESS'
--
Chain DOCKER-INGRESS (2 references)
target prot opt source destination
DNAT tcp -- 0.0.0.0/0 0.0.0.0/0 tcp dpt:3000 to:172.17.0.2:3000
RETURN all -- 0.0.0.0/0 0.0.0.0/0
```

当数据包经过宿主机的 Iptables 后，经过网桥 `docker_gwbridge` 到达命名空间 `ingress_sbox`，进行 IPVS 负载均衡后将目的地址翻译成最终容器的 IP 地址，该数据包在 ingress 网络中传输，到达最终目的地。

可见，如果要控制外部到服务的访问，可从主机的 `eth0` 接口到最终的容器之间的数据通路上部署访问控制规则，例如主机的 `iptables` 表中 `DOCKER-INGRESS` 链加 ACL，或在宿主机外部部署防火墙。当然无论哪种方式，访问控制规则都需要跟随容器的变化而更新，这就需要访问控制管理机制与容器编排体系完全集成。下图展示了外部访问 node1 的 3000 端口和 node2 的 3000 端口，不同颜色的箭头表示了数据包通过不同主机端口到容器 Web5.3 的整个数据流向。

图 4.2 Docker Swarm 集群数据流向



4.5.1.3 微分段（Micro-Segmentation）

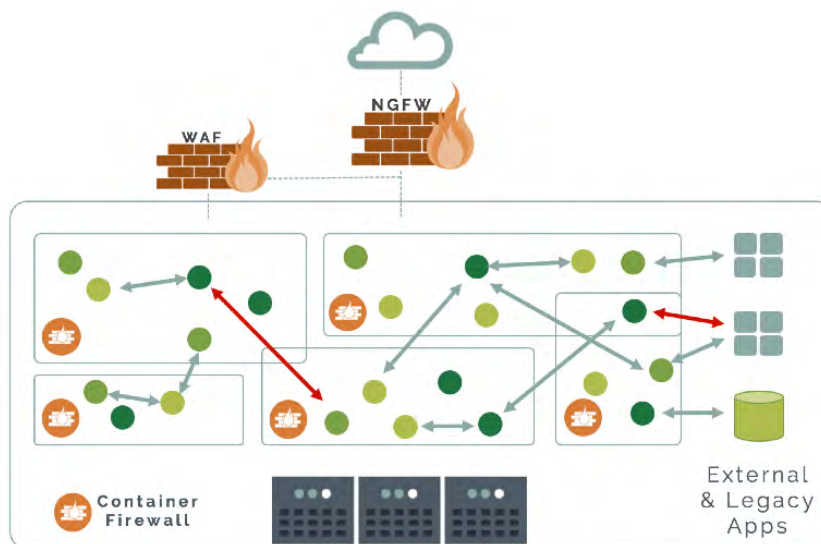
服务软件从单一的应用程序迁移为基于容器的大量微服务，在带来许多好处的同时也改变了软件内部的通讯模式。从网络和安全角度来看，最显著的变化是内部网络的东西向通信流量剧增，边界变得更加模糊。尽管每个运行中的容器都可以被加固，也可限定有限的对外网络通信接口，但因为通信接口总量的激增，也给网络攻击者探测和发现漏洞带来更多机会。

此外，容器支持秒级快速部署，容器编排系统可以根据资源情况在相同或者不同主机上自动启动新容器。每个容器都有自己的映射网络接口，这个接口可以在运行中重新分配和解除。特别是容器生命周期很短，通常 17% 的容器不到 1 分钟，78% 不到一个小时^[57]，因而容器防护需要变得敏捷弹性。

在这种动态变化的容器环境中，传统网络防火墙不仅难以看到容器之间的网络流量，而且随着容器的快速启动和消失，它也无法适应这种持续的变化。正如一位网络安全架构师所说：“在一个容器化的世界里，你无法手动配置 Iptables 或手动更新防火墙规则。”

因此在云原生的环境中，需要对应的云原生容器防火墙，它能够隔离和保护应用程序容器和服务。即使在容器动态扩展或缩小的情况下也会自动实现发现、跟随和保护。容器防火墙还可以像传统的网关防火墙一样保护从外部网络以及传统应用程序到容器环境之间的网络通信。

图 4.3 微分段中容器防火墙示意图



微分段是比传统以网络地址为粒度的分段更细的隔离机制，例如可以是单个容器、同网段的容器集合或容器应用等。微分段也是容器防火墙的基本功能之一，容器防火墙可感知第七层或者应用层，根据上层应用对连接进行动态的控制，因而容器防火墙可实现面向业务的动态微分段，成为了保护东西向流量场景中容器免受恶意攻击第一道防线。

容器防火墙主要是针对保护容器之间的网络会话，面向东西向场景，所以并不会取代部署在数据中心入口处的防火墙等系统，比如 NGFW、IDS/IPS 或 WAF。相反，容器防火墙和传统防火墙协同合作，可以有效防止源于内部应用程序级别的攻击。

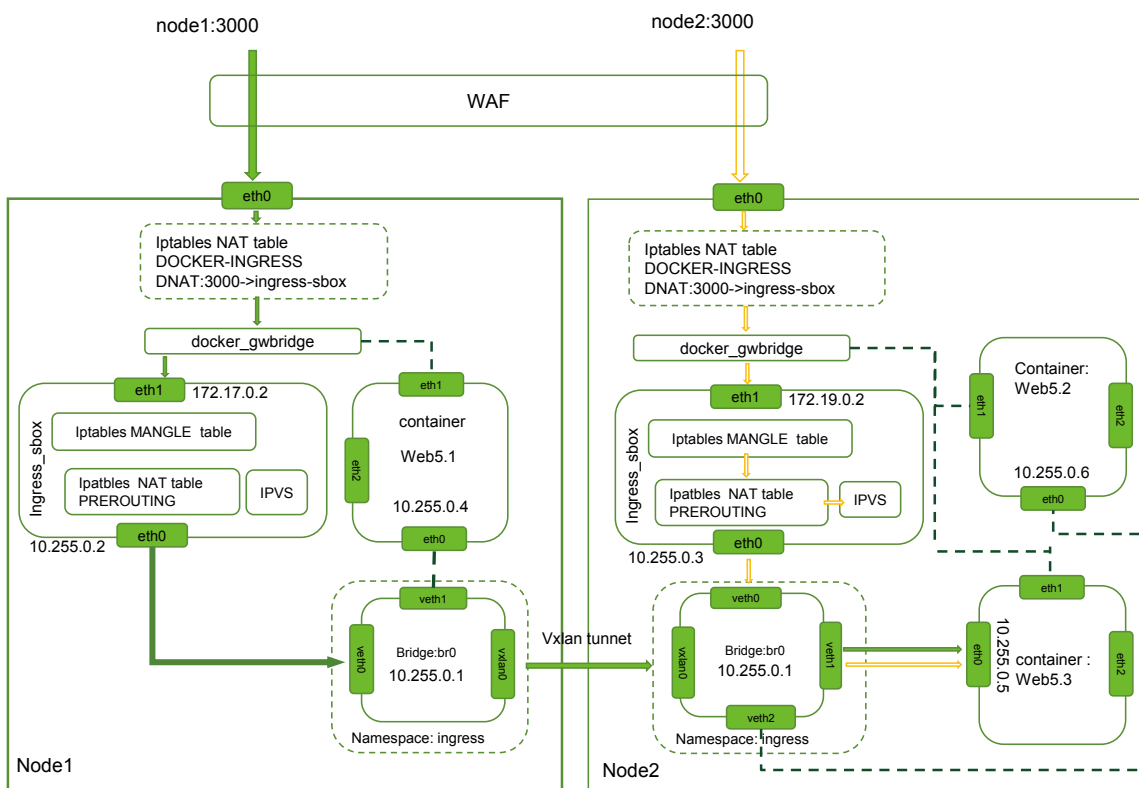
4.5.2 容器网络安全防护

4.5.2.1 微服务 Web 应用安全

对外的微服务，通常以 Web 或 HTTP 形态的应用和服务为主，如 Web 站点或支持 REST API 的应用程序。为了保障这类应用的安全，可部署 Web 应用防火墙检测并阻断恶意 HTTP 请求。本节介绍如何进行 Web 相关应用的网络安全防护，其它应用流量可见下一节威胁检测部分。

微服务接收的流量可分为南北向流量和东西向流量，以 node1 为例，从外部网络（external network）访问服务 Web5 的流量为南北向流量，该场景下可在外部网络部署 Web 应用防火墙，如图所示：

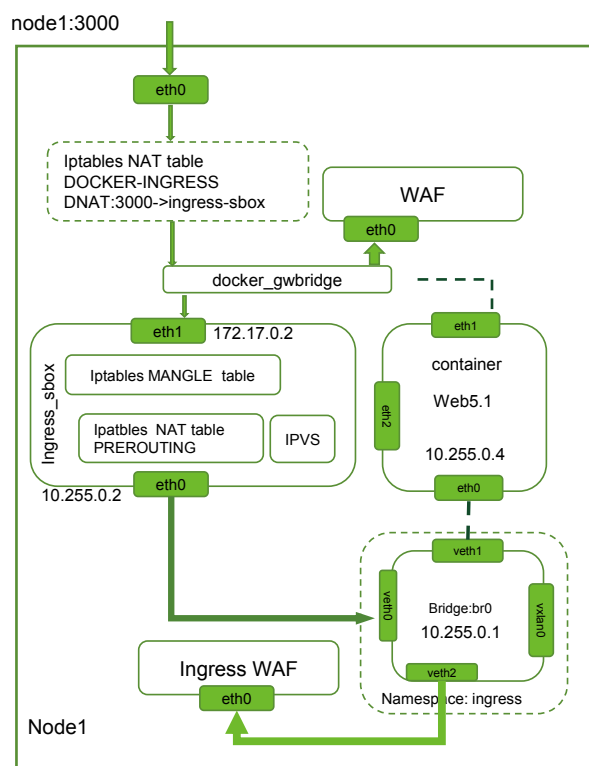
图 4.4 应用程序南北向流量防护示例



在容器环境内部，也存在大量的 API 调用，甚至可以说数据中心大部分流量均为这些 API 的调用请求和响应，这部分流量称为东西向流量。防护这些流量则需要在宿主机内部网络部署虚拟 Web 应用防火墙，形态既可为虚拟机（vWAF），也可为容器（cWAF）。由于在容器管理和编排系统中会根据业务需要，动态新增、删除容器实例，所以需要与容器管理和控制平面对接，跟随网络变化动态部署虚拟 Web 应用防火墙。

以 node1 为例，由于 Web5 连接了 Ingress 网络（10.255.0.0/24）和容器内部网络（172.17.0.0/24），所以如果要防护服务 Web5 在主机 node1 上的容器副本（本例中为容器 Web5.1），则可在网桥 docker_gwbridge 和命名空间 ingress 中的网桥 br0 旁分别部署一个虚拟 WAF，如以旁路模式镜像流量可实现恶意请求的检测，如以串接模式过滤流量可实现相应的防护。

图 4.5 应用程序东西向流量防护示例



通过软件定义网络（SDN）技术可实现流量的灵活调度，有可能使用更少的 Web 应用防火墙实现相应的安全防护，相关技术在绿盟科技的《2015 软件定义安全 SDS 白皮书》^[58] 有详细介绍，在此不做赘述。

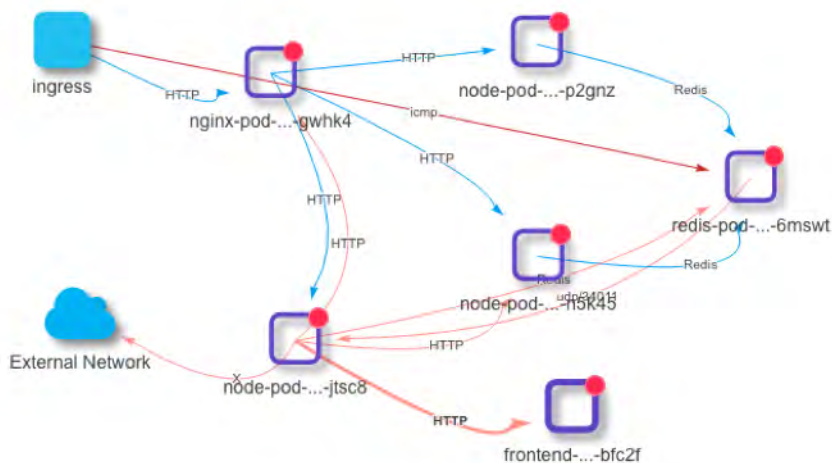
4.5.2.2 网络安全防护

在每个主机和服务服务器上部署容器防火墙，使之拥有本地 Docker Daemon 进程的完全访问权，鉴于其部署在容器网络内部，因此很容易监控到容器网络的异常行为，特别是恶意攻击者在容器间的横向移动，同时可以在内部集成机器学习模块，自动学习安全策略，从而更有效保护容器安全。

由于其分布式特性，容器防火墙可提供有效的本地监视和保护。通过与 Docker 引擎、容器编排管理工具的无缝集成，容器防火墙还可以提供除了访问控制外的其它功能，如对主机和容器的安全审计、安全测试以及资源监视。从这个意义上看，容器防火墙本质上是容器网络中的安全策略强化点，而非传统意义上的狭义防火墙。具体地，该强化点通常包含以下功能：

- 为了保护容器的部署和运行安全，容器防火墙要能够支持虚拟化的工作负载，需要能够自主学习应用程序的行为，能够智能产生安全策略以及能够与容器编排平台无缝集成。

图 4.6 容器网络通信可视化检测与显示



容器运行时，通常网络安全检测引擎需要检测如下项：

- 62

4.6 运行时安全

4.6.1 容器启动安全配置

容器以进程的形式运行在主机上，运行的容器进程是隔离的，它拥有自己的文件系统、网络以及独立于主机的进程树。下面将详细说明如何使用 `docker run`¹ 来定义容器运行时的资源。

4.6.1.1 内核选项配置

(1) 启用 AppArmor 安全配置

AppArmor 通过执行安全策略来保护 Linux 系统和应用程序免受各种威胁，这里的安全策略通过 AppArmor 配置文件体现（参考 4.1.4.2）。用户可以为容器创建自己的 AppArmor 配置文件，也可以使用 Docker 默认的 AppArmor 配置文件。

```
# docker run --security-opt="apparmor=$PROFILE"
```

(2) 设置 SELinux 安全选项

SELinux 提供了一种有效的 Linux 访问控制机制，在 Docker 服务启动时，建议对 SELinux 进行配置。那么在启动容器的时候，同样建议使用该安全选项，通过为 Docker 容器创建或导入 SELinux 策略模板，实现其安全防护。

```
# docker run --security-opt label=level:TopSecret
```

(3) 内核功能受限使用

Linux 内核的 Capabilities 机制，支持 Docker 容器内部仅授予其必要的内核功能，通过移除不必要功能，减少攻击面来增强其安全性。例如容器中通常是不需要 `NET_ADMIN`、`SYS_ADMIN`、`SYS_MODULE` 等功能的。

可以通过以下命令添加 / 删除所需的功能：

```
# docker run --cap-add={"NET_ADMIN","SYS_ADMIN"}
# docker run --cap-drop={"NET_ADMIN","SYS_ADMIN"}
```

建议在使用过程中，先删除所有的功能，并且只添加自己所需的功能：

```
# docker run --cap-drop=all --cap-add={"NET_ADMIN","SYS_ADMIN"}
```

另外，要格外注意 `--privileged` 选项，该选项可以将所有 Linux 内核功能提供给容器，而且还提升了 CGroups 控制器执行的所有限制，也就是说，此时容器几乎可以做任何主机能做的事情，因此如果不是必须的情况下，容器运行时不要使用 `--privileged` 选项。在 Linux 中，进程可以在内核中设置 `no_new_priv`，`no_new_priv` 确保进程或其子进程不通过 `suid` 或 `sgid` 位获取任何其它权限。

```
# docker run --security-opt="no-new-privileges:true"
```

(4) 不要禁用默认的 seccomp 文件

seccomp（全称 secure computing mode）是 Linux Kernel 从 2.6.23 版本开始所支持的一种安全机制。在 Linux 系统里，大量的系统调用直接暴露给用户态程序。但是，并不是所有的系统调用都被需要，而且不安全的

¹ <https://docs.docker.com/engine/reference/run/>

代码滥用系统调用会对系统造成安全威胁。通过 `seccomp`，限制程序使用某些系统调用，这样可以减少系统的暴露面，同时使程序进入一种“安全”的状态。

默认情况下，Docker 会启用 `seccomp` 配置文件。在 Docker 1.10 版本以后，默认的配置会阻止部分系统调用，即使容器已经进行了 `--cap-add` 设置，用户可以自定义该文件，通过白名单的方式获取相应的内核系统调用权限：

```
# docker run --security-opt="seccomp=profile.json"
```

（5）必要时进行 `ulimit` 设置

通常情况下，在 Docker 守护进程的配置中，会对容器的资源分配进行默认的配置，如果某些容器在资源使用上，需要定制化的配置需求，可以在启动时通过 `--ulimit` 选项对 Docker 服务的默认配置进行覆盖。

```
# docker run --ulimit nofile=1024:1024
```

（6）内核命名空间不要共享

Linux 通过内核命名空间实现相应的隔离，比如通过 PID 命名空间实现了进程之间的隔离，也就是说不同命名空间中的进程，可以拥有相同的进程 PID。如果容器共享了主机的 PID 命名空间，就意味着在容器内能够查看主机上的所有进程，甚至可以在容器内杀死主机上的进程。

对于其它的命名空间来说也是一样的，容器和主机之间如果共享了命名空间，也就失去了容器的隔离效果。因此在容器启动的时候要注意命名空间相关参数的使用，比如不要用“`docker run --pid=host`”共享主机的 PID 命名空间；不要使用“`docker run --ipc=host`”共享主机的 ipc 命名空间；不要使用“`docker run --uts=host`”共享主机的 uts 命名空间。

4.6.1.2 资源管理和控制

为了预防系统资源耗尽造成的 DoS 攻击，Docker 可以使用特定的参数进行资源限制，如限制容器的 CPU、内存或磁盘的使用量，实现资源的独占或约束。

（1）限制容器内存使用的配置

默认情况下，Docker 主机上的容器可以使用主机的所有内存资源，并不会对每个容器的可用资源进行限制。用户可以通过限制容器的内存使用量，防止由一个容器耗尽主机资源而导致拒绝服务，进而影响主机上其它容器的正常运行。

具体可使用使用“`-m`”或“`--memory`”参数运行容器：

```
# docker run --memory 256
```

（2）限制容器 CPU 使用的配置

在默认情况下，CPU 使用时间在上一主机的所有容器间平均分配，可使用 CPU 共享功能（`--cpu-shares`）来设定优先级。CPU 共享机制允许一个容器的 CPU 占用优先于另一个容器，并且禁止较低优先级的容器频繁地占用 CPU 资源。这样可确保高优先级的容器更好地运行，且可以有效的防止 CPU 资源耗尽。

比如，具体可使用使用“-c”或“--cpu-shares”参数运行容器：

```
# docker run --cpu-shares 512
```

默认情况下，每个新创建的容器都将有 1024 个 CPU 的份额（可以理解为 100%），上述例子中将 CPU 份额设置为 512，也就是说这个新创建的容器，会以其它容器 CPU 使用份额的 50% 启动。

（3）限制容器存储使用的配置

如果不对容器之间或者主机与容器之间进行数据隔离，攻击者就可以轻易获取重要数据从而造成安全风险，较好的解决方式是采用写时复制（Copy-on-Write）特性，所有运行的容器共享一个基础文件系统镜像，容器内的文件系统会与基础文件系统进行交互，这样可以避免容器间的数据共享风险，同时实现数据隔离。

另外，要严格控制容器的挂载目录，不能将 `/dev`、`/boot`、`/etc`、`/sys` 等主机的敏感目录作为容器卷进行挂载。同时在将主机设备通过 `-device` 共享给容器的时候也要特别注意，因为默认情况下，容器能够对这些设备进行读写，还可能会从主机中删除设备。因此尽量不要直接将主机设备共享给容器，如果必须要共享的话，要适当的限制共享权限。

4.6.1.3 网络配置

（1）不在生产环境的 Docker 主机上使用主机网络模式

默认的 Bridge 网络模式须通过虚拟化网络技术来创建自己的网络协议栈，与之相比，Host 网络模式由于使用了主机的本地网络堆栈，会有更好的网络性能。因而当网络性能要求非常高时，有的场景会使用主机网络模式运行容器。

但需要注意的是，容器将共享主机的网络堆栈，容器可看到主机的所有接口，这样就给了容器访问本地系统服务全部的权限，如 D-bus，因此，此模式认为是不安全的，不建议在生产环境中启用。

（2）尽量不要使用 Docker 的默认网桥 docker0

默认的 Bridge 网络模式下，Docker 主机上的所有容器均会使用 `docker0` 网桥进行连接，这种方式会引发局域网的 ARP 欺骗、嗅探、广播风暴等攻击，对这些容器造成安全威胁。不建议在使用容器时直接使用默认的 `docker0` 网桥。

建议用户使用自定义的网络进行容器之间的互连，比如使用如下命令，在 Docker 主机上创建自己的网络：

```
# docker network create --subnet 102.102.0.0/24 test
# docker run --network test ...
```

或者是利用 Docker Swarm 或者 Kubernetes 等容器集群管理平台，创建集群间的 Overlay 网络：

```
# docker network create -d overlay overlaynetwork-test
# docker run --network overlaynetwork-test ...
```

（3）只映射必要的端口

在端口使用方面要注意以下几个问题：一方面，在 Linux 默认情况下，普通用户不能占用 1024 以下的特权端口。特权端口通常用来接收和发送各种敏感和特权的数据，如果将容器端口映射到主机上的特权端口，将可能带来严重的影响。

当然，对于将 HTTP 服务绑定 80 端口，HTTPS 服务绑定 443 端口这种必须使用的，不在讨论范围。这里的特权端口包括 22（SSH 服务）、21（FTP 服务）、25（SMTP 服务）、110（POP3 邮件服务）等。

另一方面，严格审计容器镜像的 Dockerfile 文件，确保只暴露必要的端口。

（4）容器中不要运行 SSH 等不必要的服务

通常用户在使用容器的时候，很自然地会和虚拟机进行类比，比如怎么进入容器内进行调试、sshd 怎么配置等。要正确的使用容器，一定要有容器化的思维，正确的认识容器。特别是在微服务体系中，容器的本质就是一个或少数进程以及运行进程所需要的各种依赖，即运行时环境的最小集。

如果在容器中运行了 SSH、TELNET 等服务，不仅不会增加该微服务的功能，反而会带来一系列的安全威胁以及增加安全运维的复杂度。比如 SSH 服务的访问策略和安全合规性管理、各种容器的密钥以及密码管理、SSH 服务安全升级等。因此在生产环境中，容器内不应运行 SSH 等不必要的服务。

4.6.1.4 其它配置

（1）容器重启策略配置

若不设置容器尝试重新启动的策略，重启服务则会不断的尝试重启容器，极端情况可能会造成主机宕机。

如果一个容器异常终止，正确的做法应该是限制容器尝试重新启动的策略，并查找终止的原因，而不是无限地进行重启。在 Docker 下，通过设置 *on-failure* 限制容器尝试重启的次数。从安全的角度考虑，建议将尝试重启次数设置为 5。

```
# docker run --restart=on-failure:5 redis
```

（2）避免使用 root 用户来启动容器服务

CGroups 安全机制就是如果以 root 权限运行容器，容器内的 root 用户也就拥有了主机的 root 权限。要防止此类攻击，不要为单一的容器分配过多的资源即可避免此类问题。因此在运行容器时应该避免使用 root 用户，应该使用普通用户启动容器服务。

（3）检查容器运行时的状态

如果容器的镜像中没有定义 *HEALTHCHECK* 指令，建议在容器运行时添加 *health-cmd* 参数检查容器运行情况。例如：

```
# docker run --health-cmd = 'stat /etc/passwd || exit 1'
```

（4）尽可能限制容器拥有的 Linux 内核功能

对于容器里服务不必要的内核功能（如 Docker 默认的内核功能包括：chown、dac_override、fowner、kill、setgid、setuid、setpcap、net_bind_service、net_raw、sys_chroot、mknod、setfcaph 和 audit_write），可以通过 *-cap-drop* 和 *-cap-add* 来限制启动。

4.6.2 运行时安全监控与审计

监控和审计机制是用来保证服务安全性的常用手段，下面将介绍在容器环境下如何做安全监控和审计。

4.6.2.1 运行时监控

有别于传统环境，为了保证容器的稳定运行，在容器环境下需要从主机、容器实例、镜像等多个层面进行监控审计。

(1) 主机监控

主机监控基本就是传统主机监控的范围，通常可以包括主机的操作系统信息、CPU 信息、内存信息、进程信息、文件系统信息、网络状态等信息。主机的监控在运维体系中相对比较成熟，也有很多的开源工具进行实现，比如 Performance Co-Pilot¹、Icinga²、Munin³ 等，本文不再赘述。

(2) 容器实例监控

容器实例的监控通常包括主机上容器的基本信息（容器 ID、镜像名称、创建时间、状态、端口信息、容器名等），容器的 CPU 使用量、内存使用量、块设备 I/O 使用量、网络使用情况等资源使用信息。具体可使用 `docker stats` 查看相关信息，如下所示。

```
# docker stats 9d83d3116584
```

CONTAINER ID	NAME	CPU%	MEM USAGE/LIMIT	MEM%	NET I/O	BLOCK I/O	PIDS
9d83d3116584	dev_proxy_1	0.00%	14.86MiB/125.9GiB	0.01%	286MB/286MB	14.3MB / 8.19kB	25

4.6.2.2 恶意行为监控

图 4.7 多维度恶意行为监控



¹ <https://pcp.io/>

² <https://www.icinga.com/>

³ <http://munin-monitoring.org/>

(1) 容器监控

攻击者可利用应用程序的漏洞、错误配置进入容器，然后以这个容器为跳板，对内部网络进行探测，进而发现其它容器或者宿主机的进程和文件系统中的弱点来加以进一步利用和攻击。相应的保护方法是需要实时监控容器内可疑进程（如端口扫描进程和反向 Shell 进程等）。具体包括：

容器进程监控

被恶意攻击占领的容器会启动非法进程，例如数字货币挖掘软件或网络端口扫描等恶意进程，目前观察到这类恶意攻击的行为在容器环境下的趋势越来越明显。

考虑到轻量级的容器环境中，通常每个容器中运行一组数量有限且明确定义的正常进程，这些进程都是相对稳定和一致的，因而进程级别的白名单安全策略可以有效检测和阻止异常或恶意进程。

具体进程监控可以有如下几种方式。

1. 扫描 /proc 目录

/proc 是 Linux 环境下的一个虚拟文件系统。对于每一个运行的进程，在 /proc 目录下都有相对应的以进程 ID 命名的目录。目录中的一系列文件反映了运行进程的信息。比如，每一个进程的目录里面一般会有如下文件或目录：

- /proc/pid/cmdline：进程运行的命令行参数；
- /proc/pid/cwd：进程的当前运行目录；
- /proc/pid/environ：进程的环境变量；
- /proc/pid/fd：目录中包括进程运行时打开文件的文件描述符；
- /proc/pid/maps：进程使用的执行文件、库和内存的映射信息；

/proc 目录中还有其它的文件提供更多的进程相关信息。通过对 /proc 目录的扫描，我们就可以对系统和容器运行的进程有一个详细的了解。

2. 通过 netlink socket 监视进程活动

扫描 /proc 目录虽然可以获取进程的详细信息，但是如果系统中进程比较多，反复读取文件系统就会产生不必要的开销。如果能够在进程状态发生变化时，从系统中得到实时地通知，从系统资源利用和响应速度角度考虑都是一个更好的解决方案。

Linux 系统恰好提供了这样一种异步通知的机制，这一机制是通过 netlink 消息来实现的。netlink 消息是 Linux 内核与用户环境通讯的一种机制，和系统调用（syscall）、输入输出控制（ioctl）等其它通讯机制相比，netlink 对 Linux 内核的影响较小，易于开发维护；用户层的编程界面使用的是 socket 的流程，而且提供了广播和组播的功能，多个进程可以同时获取内核的状态变化。这些都是其它通讯机制难以比拟的优势。

netlink socket 使用的是 AF_NETLINK 地址族。若要获取进程信息，需要在打开 socket 的时候指定 NETLINK_CONNECTOR 协议，然后发一个 netlink 消息订阅进程活动报告即可。通过指定不同的协议号，用户层程序还可以监视内核中路由、ARP 和 iptables 等关键组件的行为。

3. 使用系统性能分析工具 perf 获取进程活动信息

Perf event 最初的用途是为了调试和诊断 Linux 内核及应用程序的性能。它提供了极为广泛的系统行为参数的汇报和统计。其中包括系统硬件事件，比如 CPU 时钟、流水线指令执行、中断等事件的分析和软件行为，比如进程切换、内存、文件和 socket 操作等事件的报告。

对于进程有关的事件，perf event 提供了 PERF_RECORD_FORK，PERF_RECORD_EXIT 事件的记录报告。Perf event 的报告既可以获取事件计数的统计值，又可以通过 mmap 分配的共享内存使用取样方式读取事件的详细信息。在使用 perf event 获取系统信息的时候，应当事先对事件在常态和极端条件下（比如被攻击的时候）发生的频率进行详细的分析，尽量避免漏报或资源溢出的情况发生。

4. eBPF

eBPF 的全称是 extended Berkeley Packet Filter。它的强大之处在于允许用户以一定语法书写的程序可以在运行时放入内核，在指定事件发生时触发运行。内核对放入的程序进行检查，确保不会产生死循环和越界操作。设计之初 eBPF 只是用来过滤网络包，但是由于其功能强大、安全高速，在新的内核中 eBPF 的功能逐渐增强。使用 eBPF 不但可以控制网络流，还可以过滤系统调用，提高系统监测调试的性能。

eBPF 可以与 perf event 或者 netlink connector 协同使用，监控系统进程的活动。这不但大幅度提高了检测软件的性能，也使得准确度得以提升

容器文件系统监控

攻击者可以伪装或者利用开源软件库 / 软件包，开发者在不经验证导入后可被容器进程加载，这些恶意软件包甚至可以篡改容器系统的敏感文件。一旦成功，就可以尝试将自身权限升级到 root 来进行进一步的感染。例如，近期发现的 SSH 开源库中包含偷窃用户密码的恶意代码，该开源库多年来已经被大量应用在很多商用软件和容器当中。

因而，容器安全系统需要实时监控文件系统的变化，保证重要系统文件的一致性。此外应自动触发漏洞扫描并生成警报。

容器数据盗窃

攻击者通常通过一系列的技术窃取容器中的数据，这些技术包括建立反向 Shell 来接收和挖掘数据、加密和隐藏发送机密数据等。传统的 DLP 技术无法有效防护容器数据安全，安全系统需要使用新一代容器 DLP 技术、实时监控容器的行为，甚至分析攻击链事件异常来侦测此类数据窃取。

(2) 主机监控

主机监控是容器安全中的重要部分，因为运行容器的主机同样存在影响容器安全的风险，比如权限提升、突破容器主机隔离等攻击。此外，容器中的可疑行为也有一定机会在主机被检测到。例如，如果容器中的端口扫描或反向 Shell 进程开始运行，主机安全系统应该能检测到网络和进程行为异常，可实时告警响应。

运行容器的主机本身可能会受到各种新型的来自容器层面的攻击。例如，DirtyCow^[59] 的 Linux 内核漏洞使用户能够从容器升级到主机的 root 权限。进而从容器逃逸到主机节点，严重影响整个容器运行环境安全。

(3) 镜像监控

作为容器的运行基础，还需要对主机上保存的容器镜像进行监控，防止镜像被恶意篡改（比如修改暴露端口、

病毒木马植入等)。容器镜像的监控信息一般为静态信息,通常包括镜像数量、镜像 ID、镜像名称、版本、大小、构建历史信息以及暴露端口等。可以通过 `docker images inspect containerid` 进行查看镜像当前信息,再与历史信息进行对比寻找可疑变化。

```
# docker image inspect c8c29d842c09
[
  {
    "Id": "sha256:c8c29d842c09d6c61f537843808e01c0af4079e9e74079616f57dfcfa91d4e25",
    "RepoTags": [
      "nginx:1.9"
    ],
    "RepoDigests": [
      "nginx@sha256:54313b5c376892d55205f13d620bc3dcccc8e70e596d083953f95e94f071f6db"
    ],
  }
]
```

(4) 系统容器的监控

容器平台拥有大量的辅助 / 管理容器实例,这些容器是特殊的系统容器,拥有更高的权限以及更多、更复杂的通信方式和行为。系统容器本身的安全性也十分重要。2018 年初发现的 Tesla^[60] 容器安全事件,就是 Kubernetes 的系统管理界面被黑客入侵,从而被利用部署埋藏了数字货币挖矿的容器来牟利,而且导致 Kubernetes 管理的 Tesla 的技术数据被大量暴露。所以容器安全系统还需要监控系统容器的网络以及行为。

4.6.2.3 配置审计

在 4.6.1 节中介绍了在容器启动时建议的一些安全配置,那么需要在容器运行时对这些信息进行监控、审计,确保其安全性。

`docker inspect` 命令可以查看镜像或容器实例的底层详细信息,以此来了解镜像或容器的完整构建信息,用以实现在安全上的审计功能。具体包括了基础配置,主机配置、网络配置、状态信息等。

比如可以通过采用如下命令,审计容器暴露端口情况:

```
# docker ps --quiet | xargs docker inspect --format '{{.Name}}:Ports = {{.NetworkSettings.Ports}}'
/floodlight.1.xqcd9jio5p0j4jax56a0sz5uc:Ports = map[4200/tcp:[] 6633/tcp:[] 6655/tcp:[] 8081/tcp:[] 9001/tcp:[]]
```

另外,可以借助一些开源的配置核查项目来进行配置检查,如可使用 `docker-bench-security` (详见 5.1.5 节)来进行配置参数核查。

4.6.2.4 镜像脆弱性评估

从镜像的整个生命周期来看,包括“镜像构建”→“上传至镜像仓库”→“获取镜像到本地主机”→“运行容器实例”。那么在本地的容器主机,就包含了镜像和容器实例,因此在运行中需要对本地图像和运行的容器进行脆弱性评估,确保其安全性。

镜像的脆弱性评估手段可以参见 4.4.3 节的镜像扫描，Docker 镜像扫描器数据库存储镜像 BOM 的细节和每个组件的漏洞状态。当一个新的漏洞报告给 CVE 数据库，Docker 镜像扫描器将会在数据库中查看哪个镜像相关的包会受到影响然后推送通知管理员，以便管理员及时对镜像进行修补更新。容器实例的脆弱性评估手段可以借助于云环境中使用的扫描器进行。

4.7 编排安全

容器技术的成熟推动着微服务的发展和落地，越来越多的企业开始采用微服务架构来组建自己的应用，其中，容器编排工具管理着承载各类服务的容器集群。可以说，在各类采用微服务架构的项目中，容器编排工具支撑了其核心业务。接下来以社区热度最高的编排工具 Kubernetes 为例，说明目前编排工具需要做的安全防护措施。

4.7.1 Kubernetes 安全防护

无论是 Kubernetes 社区还是第三方安全机构均针对 Kubernetes 中组件和资源的安全进行了相应改善和安全加固，以下为主要改进措施：

4.7.1.1 计算资源安全

Kubernetes 社区提供 PodSecurityPolicy^[61]，在 Kubernetes 中 PodSecurityPolicy 是一个集群级别的资源，用于控制 Pod 规范以及管理 Pod 安全，PodSecurityPolicy 对象定义了一组必须运行的条件以及相关字段的默认值，集群管理员通过 PodSecurityPolicy 可以控制以下内容。

表 4.1 PodSecurityPolicy 对应字段名称

控制方面	字段名
特权模式运行	privileged
使用 root 命名空间	hostPID hostIPC
使用主机网络和端口	hostNetwork hostPorts
使用卷类型	Volumes
使用主机文件系统	allowedHostPaths
FlexVolume 驱动程序白名单	allowedFlexVolumes
分配拥有 Pods 卷的 FSGroup	fsGroup
要求 root 文件系统为只读	readOnlyRootFilesystem
容器的用户和组 ID	runAsUser supplementalGroups
限制 root 特权升级	allowPrivilegeEscalationdefault AllowPrivilegeEscalation
Linux 功能	defaultAddCapabilities requiredDropCapabilities allowedCapabilities
容器的 SELinux	SELinux
容器使用的 AppArmor 配置文件	annotations
容器使用的 Seccomp 配置文件	annotations
容器使用的 Sysctl 配置文件	annotations

4.7.1.2 集群安全

(1) 控制 Kubernetes API 的访问

Kubernetes 的 API Server 针对所有 API 流量使用了安全传输协议 (TLS) 以确保服务间访问的安全性, 另外又提供了访问认证、访问授权、Admission Control 等访问控制机制。

(2) 控制对 Kubelet 的访问

Kubelet 的 HTTPS 端点公开了 API, 这些 API 授予了节点上容器强大的控制权, 默认情况下 Kubelet 允许对此 API 进行未经认证的访问。生产环境集群中建议开启 Kubelet 身份验证和授权。

(3) 控制集群运行时工作负载和用户的能力

- 限制集群中的资源使用

资源配额 (Resource Quota) 限制了授予命名空间的资源数量, 通常是用于限制命名空间可以分配的 CPU、内存或持久性磁盘的数量, 或是通过控制命名空间中 Pod、Service、Volume 的数量来限制资源使用。

- 控制 root 权限运行容器

Pod 在 yaml 文件中的定义包含一个安全的上下文 (security context), 用于描述此 Pod 请求访问某个节点上的特定用户 (如 root)、获得特权或访问主机所在网络以及 Pod 在主机节点上不受约束的运行其它控件。而 Kubernetes 的 Pod 安全策略 (PodSecurityPolicy) 可以限制用户或服务账户 (Service Account) 提供安全上下文设置, 比如 Pod 的安全策略可以限制卷的挂载等。

- 限制网络访问

首先 Kubernetes 中基于命名空间的网络策略 (Network Policy) 允许用户限制集群中其它命名空间的哪些 Pod 可以访问它们命名空间内的 Pod 和端口, 另外通过配额和限制范围的方法也可以控制用户是否可以请求节点端口和负载均衡服务, 最后基于插件式的控制网络规则方法可以增加额外的保护措施, 比如节点防火墙、物理分离集群节点等。

- 控制 Pod 访问

默认情况下, 对在节点上访问 Pod 没做任何限制, 但 Kubernetes 为用户提供了一组丰富的策略用于控制 Pod 运行节点的位置, 这些策略典型的有 NodeSelector, Affinity 和 Anti-affinity 等。

(4) 保护集群中各组件

- 限制对 Etcd 的访问

对于 API 来说, 拥有 Etcd 服务器后端的写权限相当于获得了整个集群的 root 权限。在 API Server 访问 Etcd 服务器时, 管理员应该使用广受信任的凭证, 比如通过 TLS 客户端证书的相互认证。通常建议将 Etcd 服务器隔离到只有 API Server 可以访问的防火墙后面。

- 启用审计日志

Kubernetes 启用审计日志来记录 API 发生破坏时进行后续的分析操作。

- 限制对 Alpha 或 Beta 的访问

Kubernetes 的 α 和 β 特性还处在积极开发阶段，所以可能会有限制或是 Bug，从而导致安全漏洞，所以在使用时可以禁用那些不需要的特性。

- 实时关注安全更新和漏洞报告

加入 Kubernetes-announce 组^[62]，能够获得有关安全公告的邮件，有关如何报告漏洞相关信息，可以参考安全报告页面^[63]。

- 对 Etcd 数据进行加密

一般情况下，Etcd 数据库包含了通过 Kubernetes API 可以访问到的所有信息，并且可以给予攻击者对集群状态的可见性。所以要使用经过良好审查的备份和加密解决方案来进行数据加密，并且考虑在可能的情况下使用全磁盘加密。

Kubernetes 1.7 包含了静态敏感信息加密（encryption at rest），它是一个 α 特性，会加密 Etcd 里面的 Secret 资源，以防止某一方通过查看这些 Secret 内容获取 Etcd 备份。虽然目前只是测试阶段，但是在关键时可以提供额外防御层级。

- 开启集成前审查第三方

许多集成至 Kubernetes 的第三方插件都可以改变集群的安全配置，所以在启用第三方插件时，应当检查扩展请求的权限，Kubernetes 中的 Pod 安全策略（PodSecurityPolicy）可以提高其安全性。

- 频繁回收基础证书

Kubernetes 中一个 Secret 或凭据的寿命越短，攻击者就越难使用该凭据，所以在证书上设置短生命周期并实现自动回收证书是安全防护的一个好办法。因此使用身份验证机制时，应该设置发布令牌的可用时间，尽量缩短寿命。

4.7.1.3 小结

在使用编排工具时注意以下几个风险项。

（1）编排工具中认证和授权机制是否合规

可为编排工具的各个访问接口提供独立的权限控制策略。比如 Kubernetes 的访问接口 API Server，由于 Kubernetes 各组件通信全部由 API 驱动，控制用户访问集群以及允许用户执行哪些操作是第一道防线。类似的还有 Docker CLI。

（2）编排工具对用户输入的合法性校验是否完善

可为编排工具自定义一个过滤用户输入内容的正则列表，以确保将非法的用户内容阻挡在外。

（3）编排工具对运行时的异常处理是否全面

运行时的异常处理流程是确保工具稳定运行的必要工作，目前 Kubernetes 集群内部各组件都有其各自的日志可供查看，组件运行时异常可直接查看日志。当集群中部署的应用在集群中运行异常时，首先要查看是否是集群的问题导致，若不是则排查是否是应用本身的问题。

（4）编排工具所依赖的运行环境是否安全

外部运行环境的安全是编排工具安全的前提，可以使用针对主机系统的安全防护工具来对编排的外部运行环境进行安全加固。另外需要加固容器宿主机以缓解主机安全配置错误。更新容器引擎至最新版本避免旧版本存在漏洞对主机造成危害。最后还有一些权限及审计的问题同样需受到重视。

4.8 应用安全

容器技术的生态圈正在逐渐被夯实，容器的各细分领域也已经出现了众多解决方案，这些都为企业的容器部署工作指明了方向。在企业容器部署的基础上，围绕着容器应用的业务流程，特别是面向应用本身逻辑的微服务架构兴起，出现了对容器业务层面的应用安全的考量。

4.8.1 微服务安全

前述章节介绍了微服务的基本概念和基本架构，容器微服务架构通用组件主要由 API Server 和基于容器的微服务应用组成。这里将微服务安全防护总结为四个方面：

- API Server 的认证授权机制；
- 保证微服务架构中的服务通信过程有完整性和秘密性的安全机制；
- 微服务的接口规范；
- 微服务的高可用性。

接下来根据以上四个切入点讨论微服务的安全防护。

(1) 认证授权

当应用的架构由单体架构转型成微服务架构后，众多的微服务通过 REST API 对外暴露，会给系统带来权限控制相关的安全问题。主要有调用者的认证鉴别和 API 级别的操作权限控制问题，且这两个问题是前后有顺序的。首先是用户身份鉴定，当鉴定完毕后，对于该用户的某个具体请求是否具有该操作的权限也要进行校验。

- 认证鉴别

目前常见的认证方案主要有两种，一种是分布式 session 方案，一种是 OAuth2 Token 方案。OAuth2 Token 更适合微服务场景，因为 OAuth2 Token 有着服务端无状态、性能高、支持移动设备、与 Spring Security 高度契合的优点。OAuth2 根据不同场景主要分为授权码模式、简化模式、密码模式、客户端模式等。而常用的为密码模式和客户端模式。

- 操作权限控制

目前主要有两种方式的防护方案，一种是 Shiro^[64] 框架实现，Shiro 是一个强大的灵活开源安全框架，可以处理认证、授权、管理和密码加密等，加上其易于上手和与 Spring 的高度契合，因此被广泛推荐。

另一种是 Spring Security^[65]，由于 Spring 社区的强大，及其对 OAuth、OpenId 的支持，以及包含了 Shiro 所具备的所有功能，也被许多开发者强力推荐，但是缺点是学习曲线陡峭。

(2) 安全通信

微服务中存在着大量的应用服务，而大量的应用服务中充满了数据的内部或外部传输。出于效率原因，很多 REST API 是基于不可靠的 HTTP 通信方式，存在信息泄露、窃取和伪造等风险。对于规模化的微服务架构来说，可靠安全的 SSL/TLS 通信协议将面向用户的 API 通道做加密传输处理，则为其提供了安全保障。

（3）微服务接口的安全规范

在微服务中，各类服务的接口以 RPC 或者 REST API/Web Service 的形式对外提供，当前基于 HTTP/HTTPS 的 REST API 为主流方式。制定安全接口规范，设计各类微服务安全的 API 是保障系统应用安全的必要措施。可以依照以下规范来对 API 进行设计：

- 对发往 API 网关的请求进行有关数据包的合法性校验；
- 依据 API 白名单策略对发往 API 网关的请求进行过滤；
- 对发往 API 网关的数据格式及内容进行合法性校验；
- 对请求中包含的操作进行监听，截断敏感操作并进入授权环节；
- 限制同一用户对 API 网关的访问量。

（4）微服务的高可用性

微服务架构的中有很多服务之间存在依赖关系，且每个服务都有离线的可能，微服务数量增多，其出现故障的几率也会增大。另外网络超时、拥塞的情况也时有发生，这些都是造成服务不可用的因素。

建议从以下几个角度提高微服务的可用性。

- 设置服务调用的超时时间，以免异常的服务调用线程挤占资源；
- 为不同的业务流建立独立的线程池，使单一异常业务流造成的资源占用不会影响其它正常业务流；
- 限制并发数，限制每个服务接口的调用速率，保护核心服务的可用性；
- 对服务进行熔断设置，使异常服务能被及时得暂停；
- 完善微服务之间调用失败的回调处理，这需要根据具体业务来进行降级设计；
- 对微服务的可用性进行监控，并为异常的微服务设置异常处理流程。

4.8.2 DevOps 安全

DevOps 作为软件生命周期管理领域的方法论，已经被越来越多的 IT 企业所接受，它所带来的价值也是毋庸置疑，前文也对 DevOps 面临的安全的问题进行了阐述。本节将就如何实现 DevOps 的安全进行具体的分析。

4.8.2.1 CI/CD 安全建议

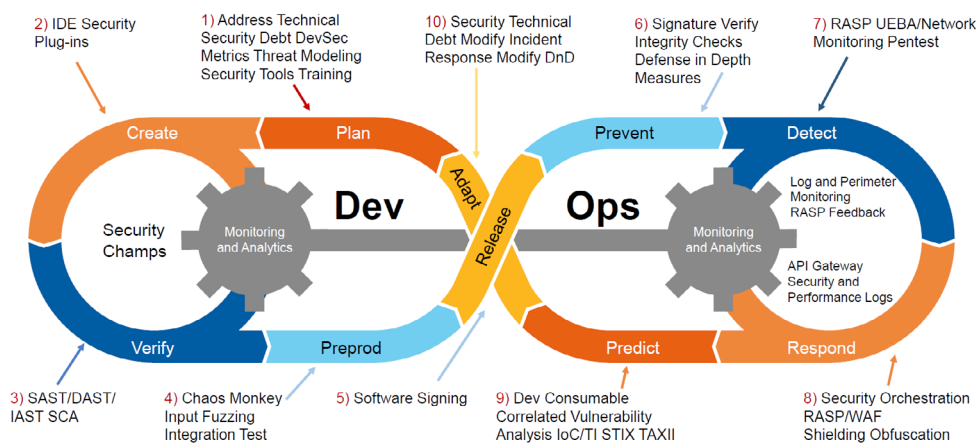
DevOps 集合了开发和运维团队，并通过自动化流程（CI/CD）来促进团队之间的协作效率并加速产品的版本迭代。以下是对 CI/CD 的一些安全性建议：

1. 将代码在隔离并受控的 Docker 容器中执行，这样会安全得多，并不会导致整个持续集成服务器受到威胁；
2. 对持续集成过程中的代码和程序进行备份，以防止环境出错导致不必要的调试操作；
3. CI/CD 流程如果涉及到有登录验证过程，那密码就有可能会有被暴力破解的风险，所以可以加上登录次数限制、手机短信验证或者第三方认证集成来验证用户的身份；
4. 在持续集成和持续部署中，对关键的测试场景和部署场景进行手动授权。

4.8.2.2 DevSecOps

提及 DevOps 安全，就不得不提到 DevSecOps^[66]，DevOps 是 Development 和 Operations 的缩写，而 DevSecOps 则是 Development、Security 和 Operations 的缩写，也就是说要在整个开发、运营的流程中，融入安全的理念。其核心是：安全是整个 IT 团队（包括开发、运维及安全团队）所有成员的责任，需要贯穿整个业务生命周期（从开发到运营）的每一个环节。

图 4.8 DevSecOps



DevOps 安全的关键是风险的控制和管理，因此在进行安全风险（Security and Risk Management, SRM）时，应当注意：

- 将安全性和合规性测试无缝集成到 DevSecOps 中，以便开发人员永远不必离开其 CI/CD 环境；
- 扫描所有开源或第三方组件中的已知漏洞和错误配置。理想情况下，构建一套完整的资产清单，来实现所有软件组件分析；
- 不要尝试删除自定义代码中的所有未知漏洞，这会增加误报。相反，将开发人员专注于那些具有最严谨和自信的人；
- 鼓励使用新类型的工具和方法，以最大限度地减少开发人员的摩擦，例如使用交互式应用程序安全测试（IAST），以取代传统的静态和动态测试；
- 使用安全冠军模型（Security Champion Model），将信息安全团队尽可能的融入到 DevOps 流程中。
- 使用统一的规范来处理所有自动化脚本、模板、图像和设计，并保证安全工作覆盖了所有的源代码。

Gartner 的调研结果显示^[67]，到 2019 年，将有超过 70% 的企业实施 DevSecOps 的相关措施，实现针对开源软件和商业软件包的自动化安全漏洞和配置扫描，而在 2016 年这一数据低于 10%。到 2021 年，80% 的快速发展团队将实践 DevSecOps，2017 年这一数字为 15%。尽管大多数情况下，DevOps 未能合理的处置安全性和合规性的问题，但值得高兴的是，企业越来越重视这一情况。

具体 DevSecOps 实施建议：

(1) 尽可能的使安全测试工具和流程适应开发人员

不要强迫 DevOps 开发者去适应传统的安全流程，相反，要尽可能的将安全集成到开发者的工具和流程中去。例如可以将安全测试工具集成到开发者的开发环境（IDE）和 CI/CD 工具链中；通过 API 构建安全和合规扫描平台，使所有的安全工具和服务都做到完全自动化；将威胁建模工具作为非功能性需求的一部分等。

(2) 不要试图在开发过程中消除所有的漏洞

没有绝对的安全，零风险是不可能的。对于 DevSecOps 而言，要做的是持续的风险和信任评估以及对应用程序漏洞进行优先级排序。实施 DevSecOps 时，信息安全人员必须接受，零漏洞既不可能也不可取，尤其是当它明显阻碍了新服务开发和创新步伐的时候。

相反，可以通过使用运行时保护的机制，来弥补已知的、较低风险的脆弱性或未知的脆弱性。例如在运行环境中，使用基于网络和主机的入侵防御系统（IPS）、Web 应用防火墙（WAF）、将应用程序性能监视和应用程序安全监控融合在一起、在应用层进行深度防御，包括负载均衡、拒绝服务和分布式拒绝服务保护等。

(3) 重点识别并移除已知的严重漏洞

通常，现代软件系统是组装而成的，而不是开发出来的。开发者们从公开可用的源代码和代码库（如 GitHub、Docker Hub）中收集并整合了大量的预编译组件、代码库、容器和框架。因此，开源软件的大量使用，对安全提出了严峻的挑战，安全扫描的焦点也有所转移。

从安全的角度来看，识别已知代码中已知的漏洞比自定义代码中的未知漏洞要容易得多。实现方式多种多样，最简单的一种方式是将文件与漏洞库相匹配。安全厂商调整它们的扫描能力，将这些操作集成到 DevSecOps 中自动化的完成，诸如 Docker 之类的安全厂商也提供此类整合能力，这种最佳实践的重要性不能低估。

(4) 增强开发人员基本的安全编码常识

DevOps 团队的每个人都应该接收安全培训，尤其是开发人员，尽可能的将安全风险在系统的源头进行规避，而不是事后的检测与处置。

例如，软件系统架构设计的安全性；在输入数据时进行适当的检测，以避免 SQL 注入和跨站脚本等攻击；数据库和配置文件以及网络流量的输入检测等。

(5) 禁止使用源代码中已知的易受攻击的组件

实际开发环境中，开发人员下载的代码库中可能包含较老的、已知的易受攻击的软件版本。为了解决这个问题，部分厂商提供了“开源软件防火墙”等工具，以便向开发人员披露这些代码库的安全态势，便于更好的决策使用哪个版本的软件。通过这种方法，开发人员可以明确的清楚那些组件和代码库存在严重漏洞，避免下载并使用。

对于那些不允许开发人员直接使用开源软件的组织，需要考虑使用安全的代码存储库模型。需要信息安全团队与开发团队协同合作，建立、审计并维护这些组织内部托管的组件存储库。

此外，一些机构会要求使用符合安全需求的、预先批准的、标准化的代码库或组件（如认证、授权、密钥管理、加密、点击劫持和输入过滤），以便于进一步降低风险。

总之，DevSecOps 更多强调的是一种理念，是一种将 DevOps 与安全进行高度集成的理念，这对传统的软件开发模式与安全管理模式是一种挑战，但其带来的安全收益也是巨大的。



2018 绿盟科技
容器安全技术报告

5. 案例介绍

5.1 Kubernetes 开源安全工具	79
5.2 NeuVector	88
5.3 StackRox	91

5.1 Kubernetes 开源安全工具

除了商业软件以外，不断发展的开源软件项目也能够提供一定的安全功能。以下是其中的一些开源项目，它们一般被用在**非关键业务**的保护场景中。

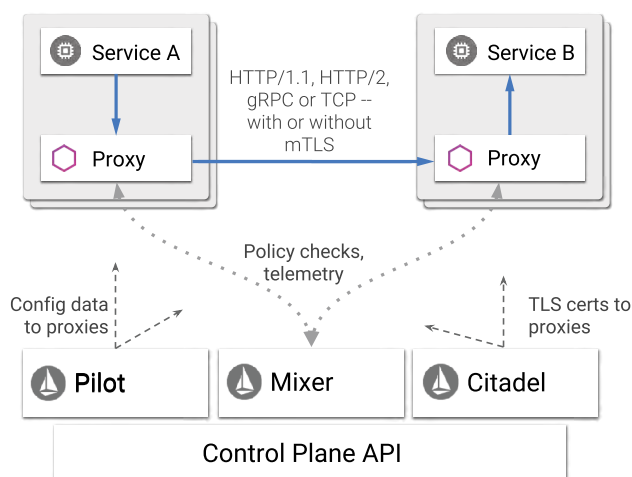
5.1.1 Kubernetes 原生网络策略

Kubernetes 网络策略提供 L3/L4（IP 地址 / 端口）级别的网络隔离。Kubernetes 本身并不提供网络功能，它只是将网络接口开放出来，进而使用相关的网络插件来支持网络策略的执行。比如 Flannel^[25]、Calico^[26]、Contiv^[68]、Cilium^[69]、Kube-router^[70] 等。

5.1.2 Istio

Istio 是由 Google、IBM 和 Lyft 开源的微服务管理、保护和监控的框架。它通过创建一个服务网格（Service Mesh）来管理服务到服务通信，包括路由、认证、授权和加密。其架构图如下：

图 5.1 Istio 架构



Istio 的定义可以简述为提供一种简单的方式来建立已部署服务的网络，具备负载均衡、服务到服务认证、监控等功能，而不需要改动任何服务代码。

如上图所示，Istio 主要由控制平面和数据平面组成：

(1) 数据平面

- Envoy：Envoy^[71] 是 C++ 开发的高性能代理，用于调解服务网格中所有服务的所有入站和出站流量。

(2) 控制平面

- Mixer：Mixer 负责在服务网格上执行访问控制和使用策略，并收集 Envoy 代理和其它服务的遥测数据。
- Pilot：负责收集和验证配置策略并将其传播到各个 Istio 组件，它从 Mixer 和 Envoy 中抽取环境特定的实现细节，为它们提供用户服务的抽象表示，独立于底层平台。
- Citadel：提供强大的服务到服务和终端用户认证，使用交互 TLS、内置身份和凭据管理方式。

Istio 同时提供了一套安全机制，秉承着不更改服务代码的原则，通过在微服务所属容器旁部署一个 Sidecar 容器¹来对服务的安全进行防护。主要分为以下两个方面：

(1) 认证

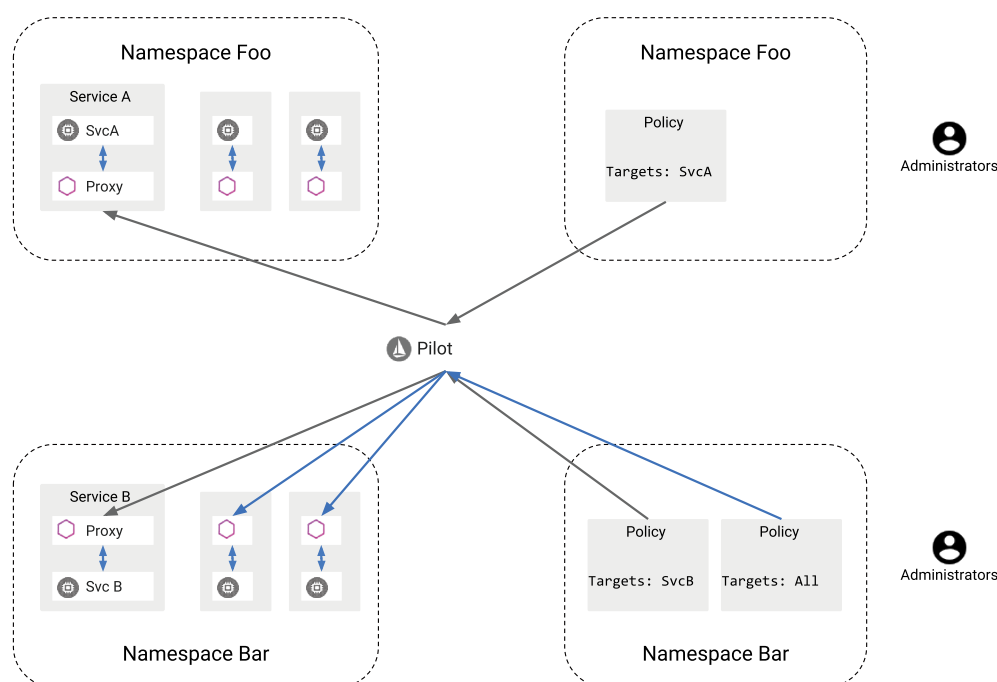
Istio 提供以下两种类型的身份验证：

- 传输身份验证（服务到服务身份验证）：Istio 提供双向 TLS 作为传输认证的完整解决方案。
- 来源身份验证（最终用户身份验证）：验证原始客户端请求作为最终用户或设备。Istio 通过 JSON Web Token (JWT) 验证和 Auth0、Firebase Auth、Google Auth 和自定义身份验证来简化开发人员体验，并且轻松实现请求级别的身份验证。

在这两种情况下，Istio 都通过自定义 Kubernetes API 将身份认证策略存储在 Istio 配置存储中。Pilot 会在适当的时候为每个代理保持最新状态以及密钥。此外，Istio 支持在许可模式下进行身份验证，以帮助用户了解策略更改在其生效之前如何影响用户的安全状态。

网络操作者可以使用身份认证策略为在 Istio 网格中接收请求的服务指定身份验证要求。认证的架构如图 5.2 所示。网络管理员（Administrators）使用 yaml 配置文件来设置策略，分别在 Namespace Foo 及 Namespace Bar 上部署，部署后策略保存至 Istio 配置存储（Istio Config Store）中，由 Pilot 监视配置存储，当有策略发生变更后，Pilot 会将新的策略转换为适当的配置并告知和服务实例位于同一位置的 Envoy Sidecar 代理如何执行所需的身份验证机制。Namespace Foo 中定义的策略目标服务为 SvcA，所以对对应策略的执行指向 SvcA 中的 Envoy Sidecar 代理。同理，下方定义的两个策略目标服务分别为 ALL 和 SvcB，所以对对应策略的执行分别指向各自的代理。

图 5.2 Istio 认证架构



¹ 一种单节点、多容器的应用设计形式。Sidecar 主张以额外的容器来扩展或增强主容器，而这个额外的容器被称为 Sidecar 容器。

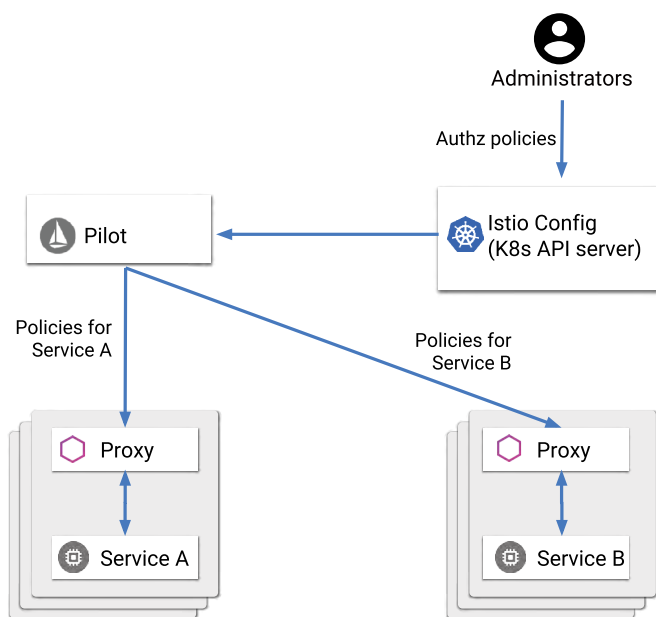
(2) 授权

Istio 的授权功能也可称为基于角色的访问控制（RBAC），RBAC 为 Istio Mesh 中的服务提供命名空间级别，服务级别和方法级别的访问控制，其特点为：

- 基于角色的语义，简单易用。
- 服务到服务和最终用户对服务的授权。
- 通过自定义属性支持的灵活性，例如条件，角色间绑定。
- 高性能，因为 Istio 授权是在 Envoy 本地强制执行的。

Istio 官方给出的授权架构如下图 5.3 所示。网络管理员可设置 yaml 配置文件的 Istio 授权策略并进行部署，部署后 Istio 将策略保存在 Istio 配置存储中，并由 Pilot 监视 Istio 授权策略的变更。如果 Pilot 发现任何更改，将获取新的授权策略。Pilot 将 Istio 授权策略分发给服务实例位于同一位置的 Envoy Sidecar 代理。每一个 Envoy Sidecar 代理都运行一个授权引擎，该引擎在运行时授权请求。当请求到达代理时，授权引擎根据授权策略评估请求上下文并返回授权结果（ALLOW 或 Deny）。

图 5.3 授权架构



5.1.3 Grafeas

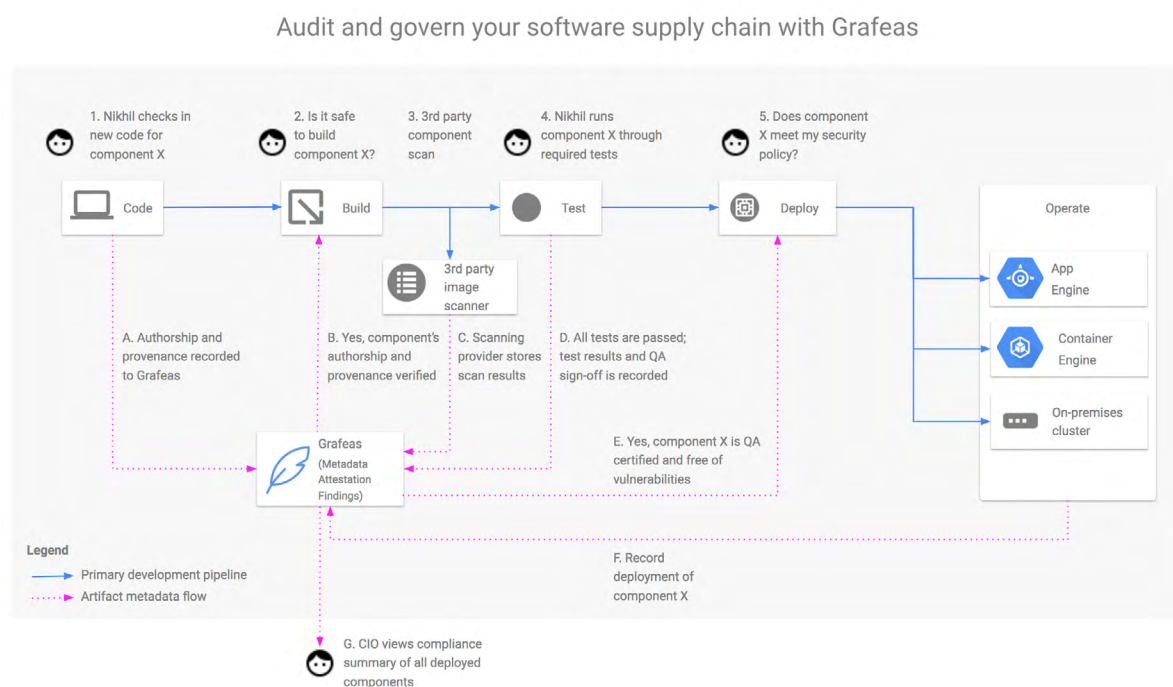
Grafeas^[72] 是 2017 年 Google 联合 JFrog、IBM 和其它一些公司推出的一款用于审计和管理软件供应链的工具。

随着 DevOps、微服务的不断发展，软件交付的速度越来越快，交付的二进制包（包括 Docker 镜像）基本呈现出指数增长的趋势。由于软件开发中多种语言的数据规范存在较大差异，例如 Go 语言使用 URL 定义自己的组件地址、Java 有自己的相应规范、Docker 使用 Sha256 表示每一层，这些规范的差异性，导致容器镜像在交付生产环境前，很难判断容器镜像内包含的多种语言的二进制包的质量信息以及漏洞信息。

Grafeas 起源于容器的安全质量管理，但从官方对其定义来看它并不局限于容器，而是通过开源工件元数据 API 提供统一方式来审计、监控软件组件。这里的“软件”可以是容器镜像，也可以是 war、jar 或 zip 包。

图 5.4 展示了在容器环境中，在镜像交付生产环境之前，多方参与的编码、构建、扫描、测试、部署步骤。在上述每个阶段，Grafeas 都会通过 Grafeas API 将其关键元数据进行收集。特别在部署阶段，可以有效地检测是否符合最终的安全规范。

图 5.4 Grafeas 审计、管理软件供应链流程图

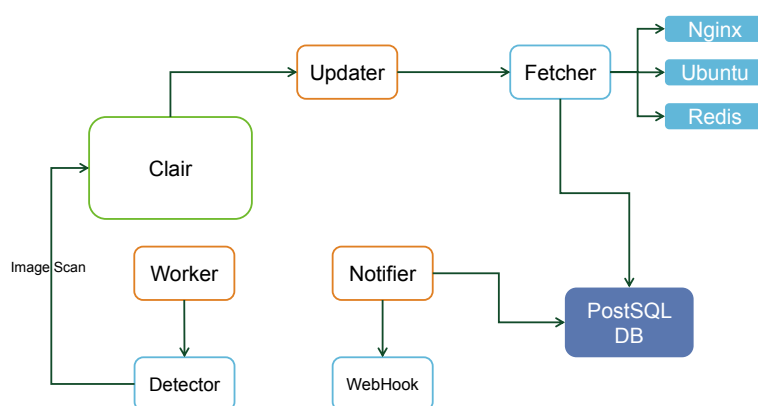


Grafeas 的安全策略可以通过与第三方工具的集成来执行，它可以用于管理 CI/CD 管道，但并不会管理运行时的容器安全。

5.1.4 Clair

Clair 是 CoreOS 于 2015 年 11 月发布的开源容器分析工具，旨在解决容器镜像的安全漏洞问题。Clair 对容器镜像执行静态分析，并将其内容与公共漏洞数据库相关联，形成对应的分析报告。

图 5.5 clair 架构

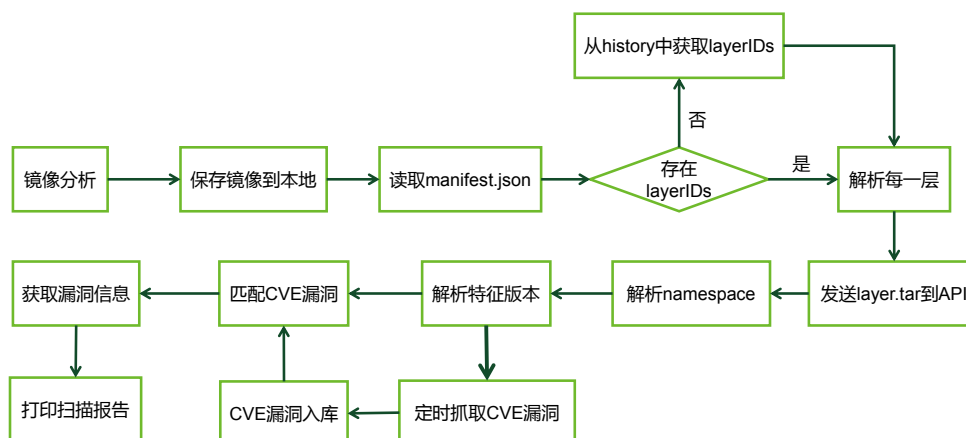


Clair 模块主要包括检测器（Detector）、获取器（Fetcher）、通知器（Notifier）和数据库（PostSQL）等。当对某个镜像进行检测时，首先对镜像进行特征的提取，然后再将这些特征匹配 CVE 漏洞库。具体流程包括：

- 用户调用 Clair 的 REST API，上传待检测镜像的每一个 layer，对应每一个 layer，Clair 都会创建一个 Worker 对其进行处理；
- 每一个 Worker 启动一个 Detector 来对收到的 layer 进行扫描；
- Detector 首先对上传的 layer 压缩包进行解压，然后解析出操作系统的类型以及版本，最后解析出其所安装应用的名称及版本；
- Updater 启动一个 Fetcher，Fetcher 根据前一步解析到的操作系统类型，查找对应的 CVE 漏洞库，查看当前版本的操作系统以及其上的应用程序是否存在漏洞。

对于漏洞信息，Clair 将会启动一个 Notifier+WebHook 的异步流程。WebHook 不断的在各大 CVE 网站爬取最新的漏洞信息，当暴露出新的 CVE 漏洞时，通知 Notifier 并将漏洞信息更新到数据库中。

图 5.6 镜像扫描流程



用户可以通过轻量的 Clair 客户端工具 Clairctl^[73] 执行简单的命令行，实现镜像的上传、扫描、输出报告等。

5.1.5 CIS Benchmarks

互联网安全中心^[74]（Center for Internet Security, CIS）是一家利用全球 IT 社区力量保护私人 and 公共组织免受网络威胁的非盈利性机构。CIS Benchmarks 被认为是保护 IT 系统和数据免受常见攻击的全球标准和公认的最佳实践。目前提供了包括操作系统、系统软件、云服务商、移动设备等众多类别的安全评测标准。

5.1.5.1 Docker CIS Benchmark

Docker 公司与 CIS 合作制作了一份安全基准文档，用于审计容器环境的安全，当前包括五个版本¹，分别是

- CIS v1.0.0 for Docker 1.11.0 Benchmark;
- CIS v1.0.0 for Docker 1.12.0 Benchmark;
- CIS v1.0.0 for Docker 1.13.0 Benchmark;
- CIS v1.0.0 for Docker 1.6 Benchmark;
- CIS v1.1.0 for Docker Community Edition Benchmark

例如 CIS v1.0.0 for Docker 1.13.0 Benchmark，该基准文档发布于 2017 年 1 月 19 日，从主机配置（Host Configuration）、Docker 服务配置（Docker Daemon Configuration）、Docker 服务配置文件（Docker Daemon Configuration Files）、容器镜像和创建文件（Container Images and Build File）、运行时安全（Container Runtime）和 Docker 安全操作（Docker Security Operations）六个方面，提出了 100 多条安全配置与审计方法。

Docker 社区将该工具进行了开源实现^[75]，当前的开源版本支持 CIS Docker Community Edition Benchmark v1.1.0。

图 5.7 Docker Bench Security 示例

```
# -----
# Docker Bench for Security v1.3.3
#
# Docker, Inc. (c) 2015-
#
# Checks for dozens of common best-practices around deploying Docker containers in production.
# Inspired by the CIS Docker Community Edition Benchmark v1.1.0.
# -----

Initializing Fri Jul 14 09:18:42 UTC 2017

[INFO] 1 - Host Configuration
[WARN] 1.1 - Ensure a separate partition for containers has been created
[NOTE] 1.2 - Ensure the container host has been Hardened
[PASS] 1.3 - Ensure Docker is up to date
[INFO] * Using 17.06.0 which is current
[INFO] * Check with your operating system vendor for support and security maintenance for Docker
[INFO] 1.4 - Ensure only trusted users are allowed to control Docker daemon
[INFO] * docker:x:992:vagrant
[WARN] 1.5 - Ensure auditing is configured for the Docker daemon
[WARN] 1.6 - Ensure auditing is configured for Docker files and directories - /var/lib/docker
[WARN] 1.7 - Ensure auditing is configured for Docker files and directories - /etc/docker
[WARN] 1.8 - Ensure auditing is configured for Docker files and directories - docker.service
[INFO] 1.9 - Ensure auditing is configured for Docker files and directories - docker.socket
[INFO] * File not found
[INFO] 1.10 - Ensure auditing is configured for Docker files and directories - /etc/default/docker
[INFO] * File not found
[INFO] 1.11 - Ensure auditing is configured for Docker files and directories - /etc/docker/daemon.json
[INFO] * File not found
[WARN] 1.12 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-containerd
[WARN] 1.13 - Ensure auditing is configured for Docker files and directories - /usr/bin/docker-runc

[INFO] 2 - Docker daemon configuration
[WARN] 2.1 - Ensure network traffic is restricted between containers on the default bridge
[PASS] 2.2 - Ensure the logging level is set to 'info'
[PASS] 2.3 - Ensure Docker is allowed to make changes to iptables
[PASS] 2.4 - Ensure insecure registries are not used
[PASS] 2.5 - Ensure aufs storage driver is not used
```

¹ 数据统计截至 2018-7-31。

Docker Hub 提供了该 Benchmark 的镜像，可以通过以下命令快速实现部署应用，Docker 版本需要在 1.13.0 及以上。

```
# docker run -it --net host --pid host --userns host --cap-add audit_control \
-e DOCKER_CONTENT_TRUST=$DOCKER_CONTENT_TRUST \
-v /var/lib:/var/lib \
-v /var/run/docker.sock:/var/run/docker.sock \
-v /usr/lib/systemd:/usr/lib/systemd \
-v /etc:/etc --label docker_bench_security \
docker/docker-bench-security
```

5.1.5.2 Kubernetes CIS Benchmark

CIS 同时还提供了 Kubernetes 的安全基准文档，用于对 Kubernetes 运行环境进行审计。当前包括两个版本，分别是 CIS v1.0.0 for Kubernetes 1.6 Benchmark 和 CIS v1.2.0 for Kubernetes 1.8 Benchmark¹。最新版本为 v1.2.0，支持 Kubernetes 1.8。

CIS v1.2.0 for Kubernetes 1.8 Benchmark 发布于 2017 年 10 月，基准文档从 API Server 配置、Scheduler 配置、Controller 配置、Etcd 等多个层面，提出了 100 多个安全审计的检测项目。

针对 Kubernetes 的合规和审计检查工具，也有众多组织对其进行了开源实现。其中包括 NeuVector 的 NeuVector kubernetes-cis-benchmark^[76]、Aqua 的 kube-bench^[77] 等（均支持 Kubernetes 1.6 和 1.8）。

图 5.8 NeuVector CIS Kubernetes Benchmark

```
# -----
# Kubernetes CIS benchmark 1.6
#
# NeuVector, Inc. (c) 2016-
#
# -----
[WARN] 1.1.1 - Ensure that the --allow-privileged argument is set to false
[WARN] 1.1.2 - Ensure that the --anonymous-auth argument is set to false
[PASS] 1.1.3 - Ensure that the --basic-auth-file argument is not set
[PASS] 1.1.4 - Ensure that the --insecure-allow-any-token argument is not set
[PASS] 1.1.5 - Ensure that the --kubelet-https argument is set to true
[PASS] 1.1.6 - Ensure that the --insecure-bind-address argument is not set
[PASS] 1.1.7 - Ensure that the --insecure-port argument is set to 0
[PASS] 1.1.8 - Ensure that the --secure-port argument is not set to 0
[WARN] 1.1.9 - Ensure that the --profiling argument is set to false
[WARN] 1.1.10 - Ensure that the --repair-malformed-updates argument is set to false
[PASS] 1.1.11 - Ensure that the admission control policy is not set to AlwaysAdmit
[WARN] 1.1.12 - Ensure that the admission control policy is set to AlwaysPullImages
[WARN] 1.1.13 - Ensure that the admission control policy is set to DenyEscalatingExec
[WARN] 1.1.14 - Ensure that the admission control policy is set to SecurityContextDeny
[PASS] 1.1.15 - Ensure that the admission control policy is set to NamespaceLifecycle
[WARN] 1.1.16 - Ensure that the --audit-log-path argument is set as appropriate
[WARN] 1.1.17 - Ensure that the --audit-log-maxage argument is set to 30 or as appropriate
[WARN] 1.1.18 - Ensure that the --audit-log-maxbackup argument is set to 10 or as appropriate
[WARN] 1.1.19 - Ensure that the --audit-log-maxsize argument is set to 100 or as appropriate
[PASS] 1.1.20 - Ensure that the --authorization-mode argument is not set to AlwaysAllow
[PASS] 1.1.21 - Ensure that the --token-auth-file parameter is not set
[WARN] 1.1.22 - Ensure that the --kubelet-certificate-authority argument is set as appropriate
[PASS] 1.1.23 - Ensure that the --kubelet-client-certificate and --kubelet-client-key arguments are set as appropriate
* kubelet-client-certificate: /etc/kubernetes/pki/apiserver-kubelet-client.crt
* kubelet-client-key: /etc/kubernetes/pki/apiserver-kubelet-client.key
[PASS] 1.1.24 - Ensure that the --service-account-lookup argument is set to true
[WARN] 1.1.25 - Ensure that the admission control policy is set to PodSecurityPolicy
```

1 数据统计截至 2018-7-31。

5.1.6 TUF & Notary

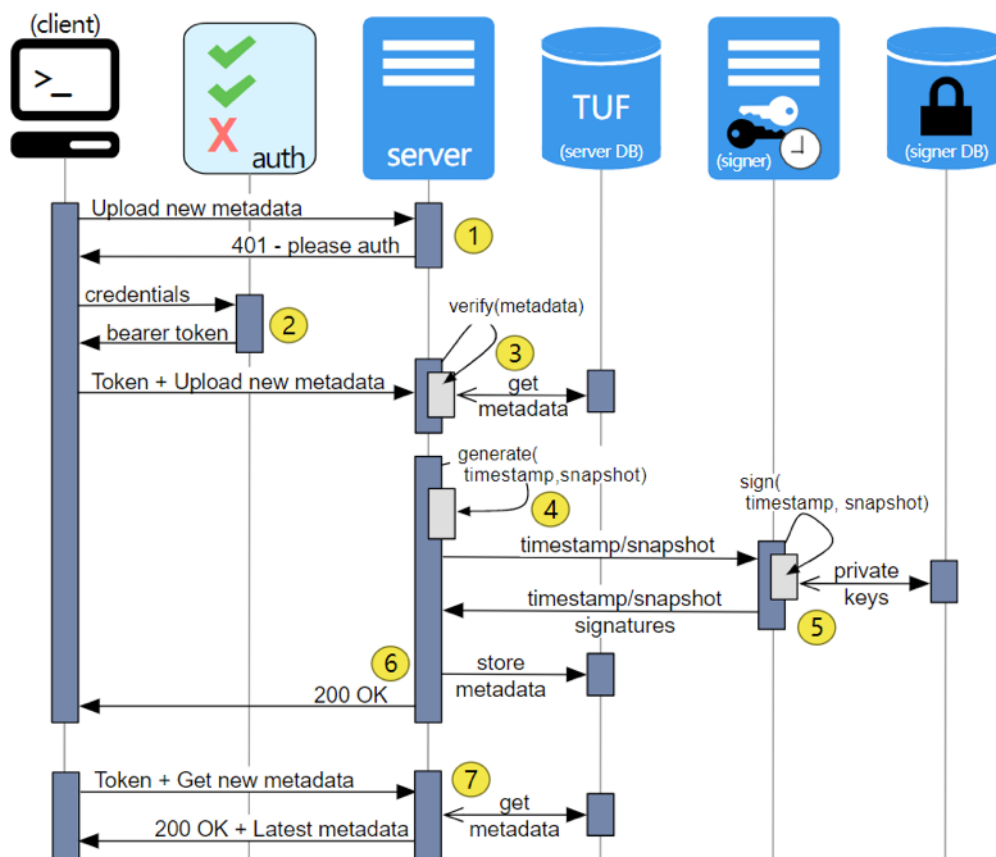
TUF^[78] (The Update Framework) 和 Notary^[79] 是云原生计算基金会^[80] (Cloud Native Computing Foundation, CNCF) 增加的两个安全孵化项目。TUF 是开源的安全和验证标准, Notary 是它的一个实现。TUF 框架最初由 Justin Cappos 教授及其在纽约大学 Tandon 工程学院的团队开发。Notary 原是 Docker1.8 版本发布的一个镜像签名系统组件。

Notary 架构主要由 Server 和 Client 组成, Server 端主要运行并维护一个容器的受信集合, Client 端与 Server 交互, 通过一种简单的方式, 实现更安全的内容发布以及内容验证。

通常, 我们可以通过 TLS 安全传输层协议来保障通信双方的安全性, 然而这种方法本质上存在缺陷, 原因很简单, Server 端的任何妥协, 都将导致恶意内容替换合法内容。

在 Notary 的帮助下, 容器镜像的发布者可以通过强安全的 Key 对容器镜像进行签名加密。一旦完成并准备发布, 镜像首先被推送至 Notary 受信的集合中。访问者只有通过安全隧道获取了发布者的公钥之后, 才能和 Notary 中的 Server 建议通信, 并访问到合适准确的镜像。

图 5.9 Notary 工作流



5.1.7 Spiffe

Spiffe^[81] 是 Google 积极推动的、开放的、社区驱动的服务安全框架，主要用于识别和保护基于 Web 的服务间通信。

当前服务的实现，通常都是以 IP 为导向，一个 IP 下面可能有非常多的东西。比如一个 IP 下面跑了很多个容器，每一个容器都做了不同的事情，怎么知道这个服务是合法的呢？又比如 A 服务是为另一个服务 B 提供服务，怎么知道 A 在真正的服务于 B 而不是做其它的任务。

Spiffe 框架定义了服务之间如何识别自身的标准，并统一实现为 URI，为生产环境中的每个对象采用 X.509 证书提供安全身份标识。

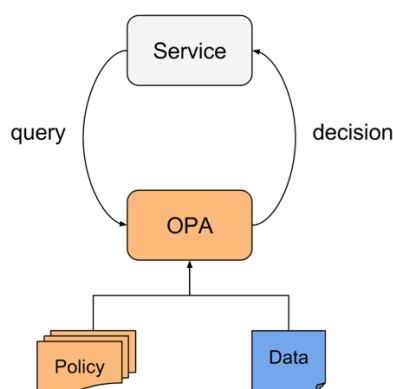
Spire^[82] 是 Spiffe 的一个开源实现。Spire 是一个工具链，用于在各种托管平台之间建立软件系统之间的信任。具体而言，Spire 公开了 Spiffe Workload API，它可以证明正在运行的软件系统并向它们发出 Spiffe ID 和 SVID。这反过来允许两个工作负载在彼此之间建立信任，例如通过建立 mTLS 连接或通过签署和验证 JWT 令牌。

Spiffe 由 CNCF 作为沙箱级项目托管。

5.1.8 Open Policy Agent (OPA)

开放策略框架^[83] (OPA) 是一个开源的通用策略引擎，OPA 指出在云原生的环境下怎样做策略执行 (Policy Enforcement)。通过 OPA 可实现统一的、有上下文感知功能的安全策略。当服务与 OPA 集成后，用户将能够制定和部署自定义的策略，以控制服务的基于策略的功能，如下图所示。OPA 由 CNCF 作为沙箱级项目托管。

图 5.10 OPA 架构示例



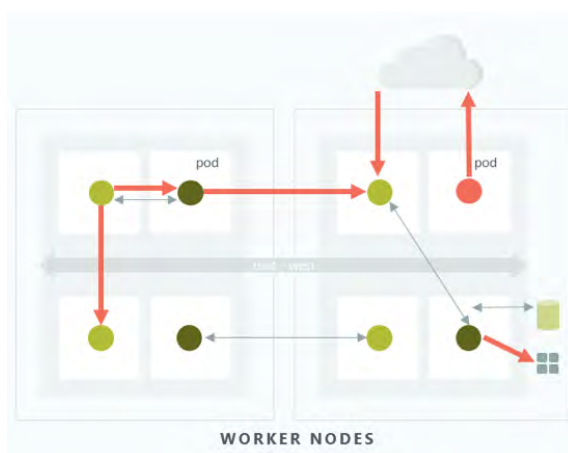
5.2 NeuVector

NeuVector^[84] 是最早开发 Docker/Kubernetes 安全产品的公司，致力于保障企业级容器平台安全，产品适用于各种云环境、跨云或者本地部署等容器生产环境。

5.2.1 云原生容器部署

NeuVector 提供实时深入的容器网络可视化、东西向容器网络监控、主动隔离和保护、容器主机安全以及容器内部安全。和容器管理平台无缝集成并且实现了应用级容器安全的自动化。

图 5.11 NeuVector 部署示意图

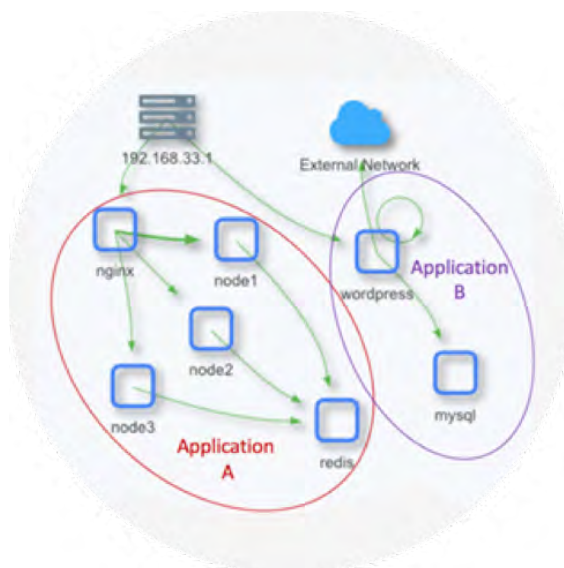


NeuVector 以云原生容器方式发行，可以通过任何标准的容器管理平台来快速部署和管理升级，比如 Kubernetes、Docker、Open Shift、Rancher、Mesosphere。同时也完全支持各种公有云、私有云容器服务平台，例如 AWS ECS、EKS、Google GKE、IBM Cloud、Docker Datacenter、Azure、Alibaba Cloud、VMware PKS 等。

5.2.2 容器环境可视化

NeuVector 能够灵活支持预先部署或者滞后部署的各种安装环境，自动发现并且即时保护已运行容器或是新部署容器。NeuVector 部署后会自动学习所有容器的网络行为以及容器内部行为。综合分析后提供实时的容器网络连接图，强大的容器可视化可以用于安全监控、拓扑分析、即时调整、事件响应、容器网络抓包甚至应用容器调试等。

图 5.12 NeuVector 容器环境可视化



5.2.3 安全审计

NeuVector 支持各种安全审计功能。提供了包括 Docker Bench 扫描、Kubernetes CIS Benchmark 扫描、容器漏洞扫描、容器主机扫描等全面安全审计报告。审计的结果可以用来制定主动响应的安全策略，保证容器主动或者被动部署时安全系统能够全自动响应检测控制以及阻止。NeuVector 支持容器镜像库扫描，提供 Jenkins 插件来帮助开发人员测试容器安全风险。

图 5.13 NeuVector 安全审计

Test numb...	Level	Message
3.19	PASS	Verify that /etc/default/docker file owne...
3.20	PASS	Verify that /etc/default/docker file perm...
4	INFO	Container Images and Build Files
4.1	WARN	Create a user for the container
	WARN	Running as root: wordpress
	WARN	Running as root: mysql
4.2	NOTE	Use trusted base images for containers
4.3	NOTE	Do not install unnecessary packages in ...
4.4	NOTE	Scan and rebuild the images to include ...
4.5	WARN	Enable Content trust for Docker
4.6	WARN	Add HEALTHCHECK instruction to the co...

NeuVector 能够辅助客户进行各种标准合规检查以及行业合规检查。包括 PCI、GDPR、HIPPA 等各行业合规标准。

5.2.4 多维度全面防护

NeuVector 从多维度在运行时实时保护容器安全。功能包括网络应用级别的容器自动隔离，根据容器行为自主学习容器安全策略，检测容器内部异常进程，异常文件系统读写以及容器内部的权限越级等恶意行为。

NeuVector 还内置了主动防护功能来防止各种流行的网络攻击，比如 SlowLoris DDoS, SQL Injection, DNS attack, SSH Heartbleed, Reverse shell 等。NeuVector 也同时通过监控容器主机的异常行为来达到全面容器生产环境的防护。多矢量多角度的安全保护同时提供了大量的事件数据，从而能展现全面的攻击链信息。

图 5.14 多维度运行时安全防护



在安全响应和管理方面，NeuVector 能够实时报警，阻断恶意行为，隔离容器或者服务，切断网络会话，自动抓取网络会话和数据包用于安全分析以及证据提取。NeuVector 支持标准的 SIEM 系统集成，支持 Webhook 以及 Syslog，支持 REST API 以及 CLI。

NeuVector 在 Docker/Kubernetes 平台之上提供高级安全功能来保障容器从开发、测试、运维到部署、升级、生产的全流程安全。

图 5.15 NeuVector 全生命周期安全防护

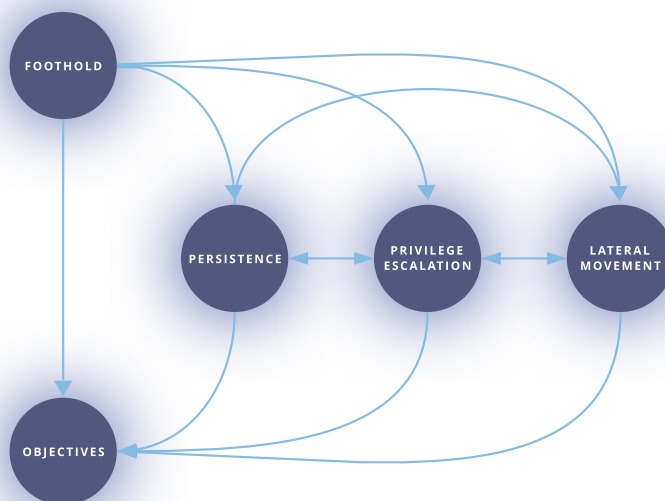
		KUBERNETES/ DOCKER	NEUVECTOR
Build, Infrastructure & Deployment	Registry Scanning, Jenkins Plug-In RBAC, Secrets Management Pod Resource & Security Management	✓ ✓ ✓	✓ RBAC Integration
	Network Policy – L3/L4 Isolation	✓	✓
Network Security	Layer 7 Deep Packet Inspection, Isolation Threat Detection Automated Packet Capture		✓
	Connection Visualization, Monitoring		✓
Endpoint Security	Suspicious Process & File System Activity		✓
	CIS Benchmarks & Run-Time Scanning		✓
Compliance Response	Automated Response – Multi-Vector		✓

5.3 StackRox

StackRox 利用分布式架构通过应用程序生命周期进行数据收集与分析，从而检测并阻止恶意攻击者，最终解决容器化云原生应用程序的保护需求。StackRox 通过独特的分布式传感器组合以及集中式分析与机器学习技术提供持续检测能力，以便帮助用户立足容器速度与规模提供情境与关联信息。

Adversarial Intent Model (AIM) 是 StackRox 提出的产品设计思路。随着现代信息技术的快速发展，传统的扫描、补丁、阻止、监控、完成这一系列独立操作的安全手段似乎不能完全适用。StackRox 提出了从攻击者角度来制定相应的防御策略，结合容器整体运行环境，以及攻击的五个周期环节，提出了多维度的威胁检测模型 AIM。

图 5.16 攻击生命周期模型



其公司产品分为 StackRox Prevent（预防检测）、StackRox Detect & Respond（监控和响应）。

5.3.1 StackRox Prevent（预防检测）

该产品是一个基于多个维度，监控和分析容器在自动化部署时的安全性和合规性的产品。通过构建容器安全状况上下文配置文件，检测其是否使用了合适的控制以及配置措施，进而从初始阶段减小攻击面。

StackRox Prevent 核心功能包括：

- Dashboard：展示整个容器集群、事件、镜像、安全配置、告警等全面的状况；
- 安全策略管理：创建、修改安全检测策略，在新漏洞公布后，重新评估更新安全策略；
- 合规检测：利用 CIS Benchmarks 对 Docker、Swarm、Kubernetes 进行基线扫描，确定当前环境配置与安全基线之间的差距；
- 产品集成：能够与容器相关的平台或工具进行友好的集成。

图 5.17 StackRox Prevent 逻辑视图

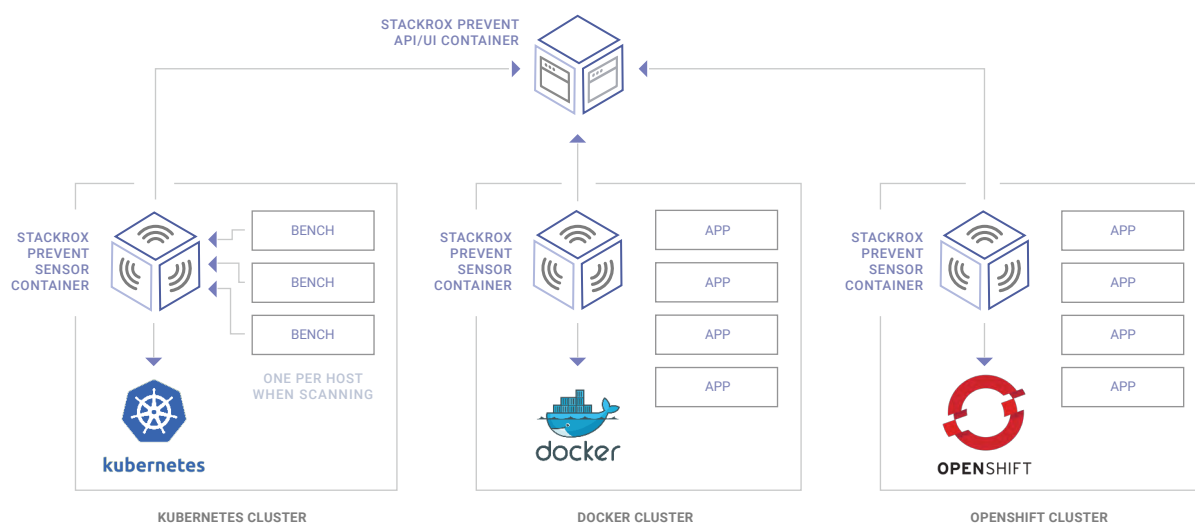


图 5.17 展示了 StackRox Prevent 的部署使用逻辑视图，主要分为感知器和处理器两个组成部分，每个组成部分以容器方式部署在容器平台或编排平台。

其中感知器（Sensor Container）分布式部署在各个容器集群中，负责检测的执行，处理器（API/UI Container）是一个逻辑上的集中控制，通过 API 与各个分布式部署的感知器进行控制和结果等信息的交互，并且通过 UI 展示给用户。

产品特点主要包括：

- 通过镜像漏洞、容器配置、主机和网络配置等丰富的环境数据，进行风险评估预测；
- 依据检测结果，快速生成综合分析报告，对风险进行量化和基准评估；
- 提供标准配置基线，同时支持自定义安全策略；
- 多种平台（DevOps、Orchestration、PaaS 等）和工具的适配支持

StackRox Prevent 目前已经完成了跟以下平台和工具的适配：

容器平台	Docker Enterprise Edition, Google Kubernetes Engine (GKE), RedHat OpenShift
IaaS	Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Bluemix, Microsoft Azure, OpenStack, Oracle Cloud, virtual machines (KVM, Hyper-V, VMware, Xen), bare metal
镜像仓库	Docker Hub, Docker Trusted Registry (DTR), Tenable, Quay
漏扫工具	Docker Security Scanning, Tenable, Quay, CoreOS Clair
身份管理	Auth0
工作流	Jira, Slack, e-mail

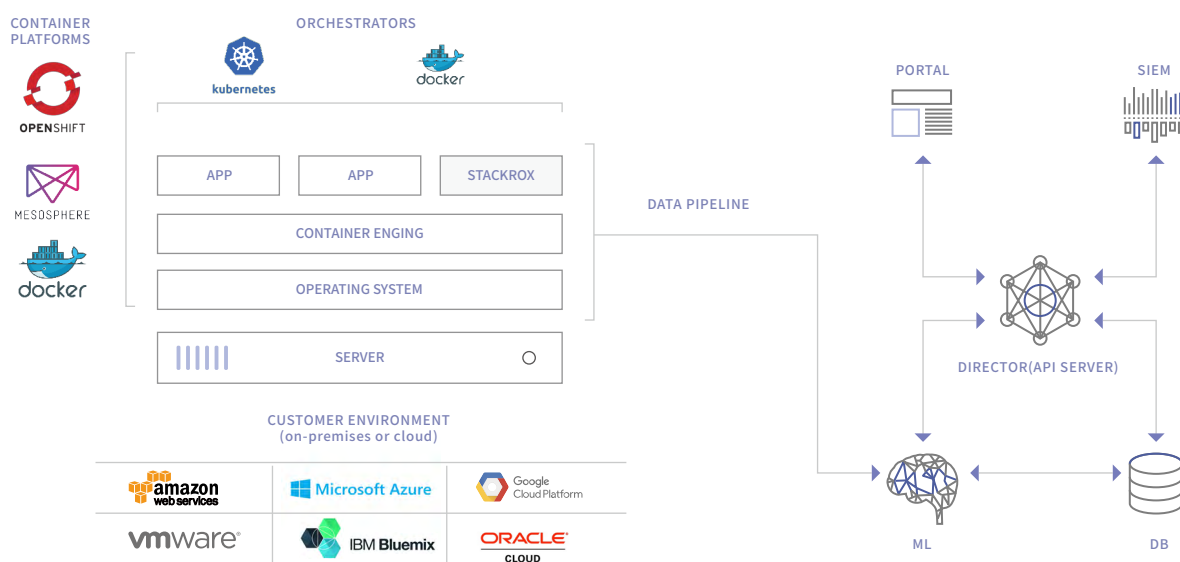
5.3.2 StackRox Detect & Respond（监控和响应）

StackRox Detect & Respond 通过与容器或云平台的集成，为本地应用程序提供内置安全性，可视化容器攻击面，展示恶意活动，通过一种新的检测和响应方式阻止攻击。

StackRox Detect & Respond 核心功能包括：

- 攻击可视化：通过部署的感知器，持续监控数百万个信号，发现每个容器在运行时存在的异常活动，并且迅速从威胁来源对其进行处理；
- 攻击链分析：通过机器学习，确定攻击者如何获取访问控制权限、如何透过业务环境，最终实施攻击，并对其根本原因进行深入分析；
- 威胁阻断：通过自动化阻断、隔离、限制威胁范围等方式，降低威胁造成的影响；

图 5.18 StackRox Detect and Respond 逻辑视图



StackRox Detect & Respond 将全新的安全架构和持续监控、机器学习相结合，保护容器环境免受新的威胁攻击，产品可以通过基于容器的微服务方式部署在任何基础设施之上。

图 5.18 展示了其逻辑视图，总体上还是分为了两部分，在架构上和 Prevent 基本一致。在用户容器集群中，通过部署 StackRox 实现数据采集、运行监控等，逻辑上集中的控制器（Director）通过收集到的数据，采用机器学习的方式进行分析判断，并做出处理决策。至于机器学习方式是如何实现的、以及其准确度，暂时还没有看到详细的说明。

产品特点主要包括：

- 深度的威胁可视化

资产可视化：发现集群中所有的容器，并且依据服务和应用对其进行分组分类，为威胁检测提供完整资产视图；

威胁可视化：低噪音的持续监控文件系统、网络、进程、以及容器事件相关内容，构成海量威胁监测点；

业务可视化：通过监控所有容器之间的通信行为，为异常监测和判断提供有效数据来源；

- 自适应的检测与响应

广泛的监测范围：目前提供了 foothold、privilege escalation、persistence、lateral movement、objectives 五个维度的监测；

自动化机器学习：根据容器活动情况的变化，完全自动化的实现学习；

自动化编排响应：通过阻断未授权指令、终止系统调用、隔离等方式，对发现的威胁进行自动化的响应处理；

策略可定制：StackRox 提供通用的预配置模板，当然用户也可以根据自身的业务 workflow，定制化自己的防护策略；

告警上下文：能够提供详细的安全事件上下文信息，以便更准确的进行威胁响应的决策；

- 快速部署、管理简单

StackRox Detect & Respond 可以直接部署在用户现有的工具链或编排系统上，和其它的容器进行统一的管理和编排。StackRox 提供 Web 界面、命令行、API 等操作方式。目前，其已经实现了以下平台和工具的适配：

容器平台	Amazon Elastic Container Service for Kubernetes (EKS), Azure Container Service (AKS), Docker Enterprise Edition, Google Kubernetes Engine (GKE), IBM Bluemix Container Service, Mesosphere DC/OS, Red Hat OpenShift
操作系统	CentOS, Debian, Red Hat Enterprise Linux (RHEL), Ubuntu
IaaS	Amazon Web Services (AWS), Google Cloud Platform (GCP), IBM Bluemix, Microsoft Azure, OpenStack, Oracle Cloud, virtual machines (KVM, Hyper-V, VMware, Xen), bare metal
镜像仓库	Amazon EC2 Container Registry (ECR), Artifactory, Azure Container Registry (ACR), Docker Hub, Docker Trusted Registry (DTR), Google Container Registry (GCR)
身份管理	SAML 2.0-compliant identity providers including Google, Okta, Ping Identity
事件告警	PagerDuty, Slack



2018 绿盟科技
容器安全技术报告

6. 总结

容器技术提供了一种轻量的虚拟化方式，在 DevOps、微服务等云原生应用的开发过程中，提供了极大的便利，经过近几年的发展，显示了强大的优势。容器最初作为开发者工具而流行，可以使用它来做隔离的开发测试环境和持续集成环境，这些都源于容器轻量、易于配置和使用的优势。

与此同时，云原生应用（Cloud Native Application）因其敏捷、易扩展、高可用等优势，被大家广泛接受。尤其是由谷歌组织成立的云原生计算基金会（CNCF），更是得到了众多厂商与开源社区的关注。

从简单的应用容器化，到云原生应用的开发，容器技术成为了其最基础也是最核心的支撑技术。新技术带来便捷与利益的同时，其安全性也需要引起足够的重视。

近年来，由于容器以及容器应用环境引发的安全风险与安全事件不断的被曝出，容器网络、容器镜像、暴露的 API、容器的隔离等问题成为了容器使用时需要着重考虑的问题。

本文从容器安全风险入手，分别从软件脆弱性、安全威胁、应用安全威胁等方面，系统的介绍了容器以及容器应用中所面临的安全问题。针对这些安全问题，从主机安全、镜像安全、网络安全、应用安全等多个角度，提出了相应的检测与防护建议。最后，从开源社区和厂商两个层面，简要介绍了当前对于容器安全的一些解决方案。



2018 绿盟科技
容器安全技术报告

7. 参考文献

- [1] Kata Containers, <https://katacontainers.io/>
- [2] Docker, <https://github.com/docker>
- [3] Rocket, <https://github.com/rkt/rkt>
- [4] Windows Containers, <https://docs.microsoft.com/en-us/virtualization/windowscontainers/about/>
- [5] Alibaba Pouch, <https://github.com/alibaba/pouch>
- [6] Tainted, crypto-mining containers pulled from Docker Hub, <https://techcrunch.com/2018/06/15/tainted-crypto-mining-containers-pulled-from-docker-hub/>
- [7] Containers At-Risk, A Review of 21,000 Cloud Environments, https://info.lacework.com/containers-at-risk-cloud-environments-review?_ga=2.217313366.1971398103.1529458438-615050303.1529458438
- [8] 国家信息安全漏洞库, <http://www.cnnvd.org.cn/web/vulnerability/queryLds.tag?pageno=3&repairLd=>
- [9] <https://docs.docker.com/enterprise/17.06/#17062-ee-6-2017-11-27>
- [10] <https://github.com/moby/moby/pull/35484>
- [11] <https://github.com/moby/moby/pull/35482>
- [12] <https://github.com/moby/moby/pull/35424>
- [13] Gartner Identifies the Top Technologies for Security in 2017, <https://www.gartner.com/newsroom/id/3744917>
- [14] Gartner Releases the Hype Cycle for Cloud Security in 2017, <https://www.gartner.com/newsroom/id/3797963>
- [15] 《Docker 容器与容器云 第 2 版》
- [16] Open Container Initiative, <https://github.com/opencontainers/image-spec/releases/tag/v1.0.0>
- [17] Docker Hub, <https://hub.docker.com/>
- [18] VMware Harbor, <https://vmware.github.io/harbor/cn/>
- [19] Docker images commandline reference, <https://docs.docker.com/engine/reference/commandline/images/>
- [20] Docker storage drivers, <https://docs.docker.com/storage/storagedriver/select-storage-driver/>
- [21] CoreOS Flannel, <https://github.com/coreos/flannel>
- [22] Calico, <https://github.com/projectcalico>
- [23] Kubernetes, <https://kubernetes.io/>
- [24] Apache Mesos, <http://mesos.apache.org/>
- [25] Docker Swarm, <https://github.com/docker/swarm>
- [26] Docker Compose, <https://github.com/docker/compose>
- [27] Rancher, <https://rancher.com/>
- [28] Docker Machine, <https://github.com/docker/machine>
- [29] 《中国云计算开源发展调查报告（2018 年）》，http://www.caict.ac.cn/kxyj/qwfb/ztbg/201804/t20180426_158539.htm
- [30] Google Kubernetes Engine <https://cloud.google.com/kubernetes-engine/>
- [31] Azure Kubernetes Service <https://azure.microsoft.com/en-us/services/container-service/>
- [32] Azure Container Instances <https://azure.microsoft.com/en-us/services/container-instances/>
- [33] Amazon Elastic Container Service <https://amazonaws-china.com/cn/ecs/>
- [34] Amazon Elastic Container Service for Kubernetes https://amazonaws-china.com/cn/eks/?nc2=h_m1
- [35] 2017 State of the Cloud Report, <https://www.rightscale.com/lp/2017-state-of-the-cloud-report>
- [36] Linkerd, <https://linkerd.io/>
- [37] Istio, <https://github.com/istio>
- [38] Kolla, <https://www.openstack.org/software/releases/queens/components/kolla>
- [39] Docker Documentation, <https://docs.docker.com/>
- [40] 国家信息安全漏洞库, <http://www.cnnvd.org.cn/index.html>
- [41] 绿盟威胁情报中心 (NTI) , <https://nti.nsfocus.com/>
- [42] 网易云镜像仓库, <https://www.163yun.com/product/repo>

- [43] 中国科学技术大学镜像仓库, <http://mirrors.ustc.edu.cn/>
- [44] DaoCloud 镜像仓库, <https://www.daocloud.io/account/signup>
- [45] 阿里巴巴开源镜像站, <https://opsx.alibaba.com/mirror>
- [46] Over 30% of Official Images in Docker Hub Contain High Priority Security Vulnerabilities, <https://www.banyanops.com/blog/analyzing-docker-hub/>
- [47] CoreOS Clair, <https://github.com/coreos/clair>
- [48] Shocker <https://github.com/gabrtv/shocker>
- [49] Gartner 2017 调研报告: DevSecOps 应当做好的十件事, <http://www.mottoin.com/107385.html>
- [50] Linux Capabilities, <http://man7.org/linux/man-pages/man7/capabilities.7.html>
- [51] Tomoyo Linux, <https://tomoyo.osdn.jp/index.html.en>
- [52] Linux AppArmor, <https://gitlab.com/apparmor/apparmor/wikis/home/>
- [53] SELinux, https://selinuxproject.org/page/Main_Page
- [54] Linux-grsec, <https://wiki.archlinux.org/index.php?title=Linux-grsec&redirect=no>
- [55] CIS Benchmarks, <https://www.cisecurity.org/cis-benchmarks/>
- [56] Docker Registry, <https://docs.docker.com/registry/>
- [57] 2018 Docker Usage Report, <https://sysdig.com/blog/2018-docker-usage-report/>
- [58] 绿盟科技 2015 软件定义安全 SDS 白皮书, <http://blog.nsfocus.net/software-defined-security-whitepaper/>
- [59] Dirty Cow, <https://dirtycow.ninja/>
- [60] Tesla, <https://arstechnica.com/information-technology/2018/02/tesla-cloud-resources-are-hacked-to-run-cryptocurrency-mining-malware/>
- [61] Kubernetes Pod Security Policies, <https://kubernetes.io/docs/concepts/policy/pod-security-policy>
- [62] Kubernetes-announce, <https://groups.google.com/forum/#!forum/kubernetes-announce>
- [63] <https://kubernetes.io/docs/reference/issues-security/security/>
- [64] Shiro, <https://shiro.apache.org/>
- [65] Spring Security, <https://spring.io/projects/spring-security>
- [66] What is DevSecOps, <https://www.redhat.com/en/topics/devops/what-is-devsecops>
- [67] 10 Things to Get Right for Successful DevSecOps, <https://www.gartner.com/doc/reprints?id=1-4I8FBLQ&ct=171016&st=sg>
- [68] Contiv, <https://github.com/contiv>
- [69] Cilium, <https://github.com/cilium/cilium>
- [70] kube-router, <https://github.com/cloudnativelabs/kube-router>
- [71] Envoy Proxy, <https://www.envoyproxy.io/>
- [72] Grafeas, <https://grafeas.io/>
- [73] Clairctl, <https://github.com/jgsquare/clairctl>
- [74] CIS, Center for Internet Security, <https://www.cisecurity.org/>
- [75] Docker CIS benchmark, <https://github.com/docker/docker-bench-security>
- [76] NeuVector Kubernetes-cis-benchmark, <https://github.com/neuvector/kubernetes-cis-benchmark>
- [77] Aqua Security, kube-bench, <https://github.com/aquasecurity/kube-bench>
- [78] The Update Framework, <https://github.com/theupdateframework/specification>
- [79] Notary, <https://github.com/theupdateframework/notary>
- [80] Cloud Native Computing Foundation, <https://www.cncf.io/>
- [81] Spiffe, <https://github.com/spiffe/spiffe>
- [82] Spire, <https://github.com/spiffe/spire>
- [83] Open Policy Agent, OPA, <https://www.openpolicyagent.org/>
- [84] NeuVector, <https://neuvector.com/>



绿盟科技创新中心

创新中心是绿盟科技的前沿技术研究部门，包括星云实验室（云安全实验室）、安全大数据分析实验室和物联网安全实验室。其作为“中关村科技园区海淀园博士后工作站分站”的重要培养单位之一，与清华大学进行博士后联合培养，科研成果已涵盖各类国家课题项目、国家标准、专利、高水平学术论文、出版专业书籍等。

创新中心持续探索信息安全领域的前沿学术方向，从实践出发，结合公司资源和先进技术，实现概念级的原型系统，进而交付产品线孵化产品并创造巨大的经济价值。

绿盟科技星云实验室

星云实验室，是绿盟科技在云计算安全方向的研究团队，团队成员来自众多国内重点院校的博士和硕士。

星云实验室的研究方向主要包括：公有云 / 政务云 / 行业云 / 私有云等云计算系统的相关安全问题以及安全解决方案的研究、SDN/NFV 等虚拟化网络安全问题的研究、Docker/Kubernetes/Service Mesh/Serverless 等云原生相关安全问题及安全解决方案研究、5G 在接入安全、切片网络安全、隐私保护等方面的安全问题以及安全解决方案研究、边缘计算在计算、存储、网络等方面的安全问题以及安全解决方案研究等新技术和新领域。

星云实验室目前已发布多份白皮书 / 专业书籍，如《2015 绿盟科技软件定义安全 SDS 白皮书》，《2016 绿盟科技软件定义安全 SDS 白皮书》、《软件定义安全：SDN/NFV 新型网络的安全揭秘》；同时参与了 CSA 云安全联盟大中华区、SDNFV 等协会组织、以及移动、联通、电信等运营商行业的标准 / 规范的制定。

星云实验室在课题研究上，承担并完成了多个国家、省、市以及行业重点单位的创新研究课题，如 863 课题、北京市科委科技课题、中国信息安全测评中心科技课题等。

星云实验室在产品落地，成功孵化落地了绿盟科技云安全解决方案，该研究内容，已由绿盟科技产品团队形成标准的产品和解决方案，为政府、金融、运营商、教育等多个行业的私有云、行业云提供了全面的云计算安全防护方案。



THE EXPERT BEHIND GIANTS 巨人背后的专家

多年以来，绿盟科技致力于安全攻防的研究，
为政府、运营商、金融、能源、互联网以及教育、医疗等行业用户，提供
具有核心竞争力的安全产品及解决方案，帮助客户实现业务的安全顺畅运行。
在这些巨人的背后，他们是备受信赖的专家。

www.nsfocus.com